Project IST-1999-11583

Malicious- and Accidental-Fault Tolerance for Internet Applications

Specification of Dependable Trusted Third Parties

Christian Cachin (editor)

IBM Research, Zurich Research Laboratory

MAFTIA deliverable D26

Public document

22 January 2001

Research Report RZ 3318, IBM Research, Zurich Research Laboratory

Editor

Christian Cachin

Contributors

Joy Algesheimer
Christian Cachin
Klaus Kursawe
Frank Petzold
Jonathan A. Poritz
Victor Shoup
Michael Waidner

Address of all authors:

IBM Research
Zurich Research Laboratory
Säumerstr. 4
CH-8803 Rüschlikon
SWITZERLAND
`http://www.zurich.ibm.com/`

Contents

1 Motivation **1**

2 Model **3**

2.1 Cryptography 5

2.2 No Timing Assumptions 6

2.3 Static Server Set 7

2.4 Related Work 8

2.5 Generalized Adversary Structures 10

2.5.1 Concept 11

2.5.2 Designing Protocols for General Adversary Structures 12

2.5.3 Differentiating Servers by Attributes 12

2.6 System Initialization 15

2.6.1 Types of Cryptographic Keys 15

2.6.2 Key Generation By Trusted Dealer 16

2.6.3 Key Generation by Distributed Protocols 17

2.7 Relation to MAFTIA Middleware Architecture 19

3 Architecture **21**

3.1 Formal Model 22

3.1.1 Basic System Model 24

3.1.2 Byzantine Agreement 33

3.1.3 Cryptographic Primitives 34

3.2 Broadcast Primitives 39

3.2.1 Reliable Broadcast 39

3.2.2 Verifiable Broadcast 45
3.2.3 Consistent Broadcast 46
3.3 Validated Byzantine Agreement 49
3.3.1 Definition 50
3.3.2 Protocols for Binary Agreement 51
3.3.3 A Protocol for Multi-valued Agreement 52
3.3.4 A Constant-round Protocol for Multi-valued Agreement 55
3.4 Atomic Broadcast 59
3.4.1 Definition 60
3.4.2 A Protocol for Atomic Broadcast 63
3.4.3 Equivalence of Byzantine Agreement and Atomic Broadcast 67
3.5 Secure Causal Atomic Broadcast 67
3.5.1 Definition 68
3.5.2 A Protocol for Secure Causal Atomic Broadcast 70

4 Applications **73**
4.1 Digital Notary Services 73
4.2 Fair Exchange TTPs 76
4.3 Certification Authority and Directory Service 80
4.4 Authentication Service 81

5 Extensions **83**

Abstract

This document describes an architecture for secure service replication in an asynchronous network like the Internet, where a malicious adversary may corrupt some servers and control the network. The underlying protocols for Byzantine agreement and for atomic broadcast rely on recent developments in threshold cryptography. These assumptions are discussed in detail and compared to related work from the last decade. A formal model using concepts from modern cryptography is developed, modular definitions for several broadcast problems are presented, including reliable, atomic, and secure causal broadcast, and protocols implementing them. Reliable broadcast is a basic primitive, also known as the Byzantine generals problem, providing agreement on a delivered message. Atomic broadcast imposes additionally a total order on all delivered messages. A randomized asynchronous atomic broadcast protocol is presented that maintains liveness and safety at the same time. It is based on a new efficient protocol for multi-valued asynchronous Byzantine agreement with an external validity condition. Secure causal broadcast extends atomic broadcast by encryption to guarantee a causal order among the delivered messages. Furthermore, it is discussed how several distributed trusted applications can be realized using such an architecture: a digital notary service, a trusted third party for fair exchange, a certification authority, and an authentication service.

1 Motivation

Distributed systems running in error-prone and adversarial environments must rely on trusted components. In today's Internet these are typically directory and authorization services like the domain name system (DNS), Kerberos, certification authorities, or secure directories accessed through LDAP. Building such centralized trusted services has turned out to be a valuable design principle for computer security because the trust in them can be leveraged to many, diverse applications that all benefit from centralized management. Often, a trusted service is implemented as the only task of an isolated and physically protected machine.

Unfortunately, centralization introduces a single point of failure. Even worse, it is increasingly difficult to protect any single system against the sort of attacks proliferating on the Internet today. One established way for enhancing the fault tolerance of centralized components is to distribute them among a set of servers and to use replication algorithms for masking faulty servers. Thus, no single server has to be trusted completely and the overall system derives its integrity from a majority of correct servers.

In this report, we describe an architecture for distributing trusted services among a set of servers that guarantees availability and integrity of the services despite some servers being under control of an attacker or failing in arbitrary malicious ways. Our system model is characterized by a static set of servers, completely asynchronous point-to-point communication, and the use of modern cryptographic techniques. Trusted applications are implemented by deterministic state machines replicated on all servers and initialized to the same state. Client requests are delivered by an atomic broadcast protocol that imposes a total order on all requests and guarantees that the servers perform the same sequence of operations; such atomic broadcast can be built from a randomized protocol to solve Byzantine agreement. We use efficient and provably secure agreement and broadcast protocols that have recently been developed.

In the first part of the report (Chapter 2), we provide a detailed discussion of these assumptions, compare them to related efforts from the last decade, and argue why we believe that these choices are adequate for trusted applications in an Internet environment. We also review the threshold failure assumptions and show how to extend them to generalized adversary structures, which allow for more adequate representation of real-world failure and trust assumptions (Section 2.5). System initialization is discussed in Section 2.6 and the relation to the MAFTIA middleware architecture [64] is described in Section 2.7.

In Chapter 3, a detailed description of our protocol architecture is given. We develop a formal model using concepts from modern cryptography, present modular definitions for several broadcast problems, including reliable, atomic, and secure causal broadcast, and present protocols implementing them. Reliable broadcast is a basic primitive, also known as the Byzantine generals problem, providing agreement on a delivered message. Atomic broadcast imposes additionally a total order on all delivered messages. We present the first randomized asynchronous atomic broadcast protocol that maintains liveness and safety at the same time. It is based on a new efficient protocol for multi-valued asynchronous Byzantine agreement with an external validity condition. Secure causal broadcast extends atomic broadcast by encryption to guarantee a causal order among the delivered messages.

We describe several possible applications of our architecture to trusted services in more detail in Chapter 4:

Certification authority and directory service: All public-key infrastructures (PKIs) use certification authorities (CAs) for binding attributes to public keys; this is done in the form of digital signatures issued by the CA. More generally, any secure directory service that uses signatures for authenticating the returned values can be realized by similar mechanisms (e.g., DNS with authentication [25]). A PKI is a typical example of a service that must be trusted and is exposed to malicious attacks on a global scale.

Fair Exchange TTPs: The problem of fair exchange of digital items or signatures over the Internet cannot be solved efficiently unless a trusted third party is used. So-called *optimistic* protocols [2] avoid calling the third party in regular executions where no faults occur. They rely on the trusted party only to recover from problems. We give the protocols for replicating a trusted third party for fair exchange in a distributed system.

Notary service: We describe a simple digital notary and time-stamping service that acts as a secure document registry with a logical clock; it assigns a sequence number to all submitted documents in the order of submission. Such a service is necessary for fully digitalizing legal processes such as filing patents. It must also prevent preliminary disclosure of the document's content before it is properly registered, as the process of filing a patent application illustrates.

Authentication service: An authentication service has to verify the claimed identity of a user. The user must present secret information that identifies her. If verification succeeds, the service will take some action to grant the request, like establishing a session or replying with a cryptographic token. Often, the answer contains a freshly generated, random session key as in Kerberos; such an authentication server is also called a key distribution center (KDC). The randomness is generated by a cryptographically strong pseudorandom generator from a secret seed.

The report concludes in Chapter 5 with mentioning possible future extensions of the model and the protocols.

2 Model

The system model described here is used for describing dependable trusted third-party services in this report. It has to be seen in relation to the general model of the MAFTIA middleware, described in a companion report [64]. Compared to the general middleware architecture, the model used here makes many simplifications (asynchrony, full connectivity, no particular network topology). On the other hand, it uses a much more formal approach to the system model, in particular in Chapter 3 for describing the architecture and the secure protocols; this is necessary for reasoning about a secure distributed system. Our approach combines the formal models of cryptography and distributed systems.

Our system consists of a static set of n servers, of which up to t may fail in completely arbitrary ways, and an unknown number of possibly faulty clients. All parties are linked by asynchronous point-to-point communication channels. Without loss of generality we assume that all faulty parties are controlled by a single adversary, who also controls the communication links and the internal clocks of all servers.

Faulty parties are called *corrupted*, the remaining ones are called *honest*. Thus, any statement about the common state of the system can rely only on the honest parties, and they proceed only to the extent that the adversary delivers messages faithfully. In short, *the network is the adversary.*

Furthermore, there is a trusted dealer that generates and distributes secret values to all servers once and for all, when the system is initialized. The system can process a practically unlimited number of requests afterwards. Sometimes, it is possible to bootstrap security from a public-key infrastructure, e.g., to establish secure point-to-point channels. Since we use specialized key generation protocols and our goal is to protect the heart of the PKI itself, however, an external mechanism is needed.

This model falls under the impossibility result of Fischer, Lynch, and Paterson [27] of reaching consensus by deterministic protocols. Many developers of practical systems seem to have avoided this model in the past for that reason and have built systems that are weaker than consensus and Byzantine agreement. However, Byzantine agreement *can be solved by randomization* in an expected constant number of rounds only (see [14] and also the survey of Chor and Dwork [18]). Although the first randomized agreement protocols were more of theoretical interest, their practical relevance has been recognized by now. For example, Guerraoui et al. [36] argue that solving consensus is central for building asynchronous distributed systems tolerating crash failures; we pursue the same approach in the model with Byzantine faults.

The recent Byzantine agreement protocol of Cachin, Kursawe, and Shoup [12] is based on modern, efficient cryptographic techniques with provable security, withstands the maximal possible corruption, and is also quite practical given current processor speed. (Its security proof uses the random oracle model, see below.)

In our architecture we use Byzantine agreement as a primitive for implementing atomic broadcast, which in turn guarantees a total ordering of all delivered messages. Note that atomic broadcast is equivalent to Byzantine agreement in our system model [16] and thus considerably more expensive than reliable broadcast, which only provides integrity of the delivered messages, but no ordering (see Chapter 3).

In the first three subsections of this chapter, we elaborate on the three key features of our model (cryptography, asynchronous communication, static server set) and then compare it to related efforts.

Of course, distributing a central service to a set of servers enhances its fault tolerance only if there is enough diversity in that set such that common failure modes can be ruled out. For example, if the same simple attack succeeds for all servers, not much has been gained by distribution. It is thus crucial for this approach to make sense that the servers vary in their configuration, operating system, physical location, load etc. Placing them in different administrative domains also eliminates corruptible system administrators as one path of attack.

The traditional assumption in distributed systems is that at most a certain fraction of homogeneous nodes fails. Based on recent progress in distributed systems and cryptography, we introduce in Section 2.5 systems that tolerate a family of novel failure patterns, which allow for more realistic modeling of real-world trust assumptions. For example, they allow a distributed system running at multiple sites to continue operating safely even if all hosts at one site are unavailable or corrupted, no matter how many there are. One may construct a distributed system that maintains its integrity despite the corruption of a majority of its servers in this way.

In Section 2.6 we address the initialization and setup in our system architecture. Although this can be done by a trusted dealer, we are interested in secure distributed protocols for this purpose to eliminate the dealer as a single point of failure.

The architecture described in this report is a special case of the general service and protocol architecture of the MAFTIA middleware, as described in the companion report [64]. The relation between these two is discussed in Section 2.7.

2.1 Cryptography

Cryptographic techniques such as public-key encryption schemes and digital signatures are crucial already for many existing secure services. For distributing trusted services, we also need distributed variants of them from *threshold cryptography* [20].

Threshold cryptographic schemes are non-trivial extensions of the classical concept of secret sharing in cryptography. Secret sharing allows a group of n parties to share a secret such that t or fewer of them have no information about it, but $t+1$ or more can uniquely reconstruct it. However, one cannot simply share the secret key of a cryptosystem and reconstruct it for decrypting a message because as soon as a single corrupted party knows the key, the cryptosystem becomes completely insecure and unusable.

A *threshold public-key cryptosystem* looks similar to an ordinary public-key cryptosystem with distributed decryption. There is a single public key for encryption, but each party holds a *key share* for decryption (all keys were generated by a trusted dealer). A party may process a decryption request for a particular ciphertext and output a decryption share together with a proof of its validity. Given a ciphertext resulting from encrypting some message and more than t valid decryption shares for that ciphertext, it is easy to recover the message; this property is called *robustness*. The scheme must also be secure against *adaptive chosen-ciphertext attacks* in order to be useful for all conceivable applications (see [58] for background information). The formal security definition can be found in the literature [60]; essentially, it ensures that the adversary cannot obtain any meaningful information from a ciphertext unless she has obtained a corresponding decryption share from at least one honest party.

In a *threshold signature scheme*, each party holds a *share* of the secret signing key and may generate shares of signatures on individual messages upon request. The validity of a signature share can be verified for each party. From $t+1$ valid signature shares, one can generate a digital signature on the message that can later be verified using the single, publicly known signature verification key. In a secure threshold signature scheme, it is infeasible for a computationally bounded adversary to produce $t+1$ valid signature shares that cannot be combined to a valid signature (robustness), and to output a valid signature on a message for which *no* honest party generated a signature share (no forgery).

Another important cryptographic algorithm is the *threshold coin-tossing scheme* in the randomized Byzantine agreement protocol of Cachin, Kursawe, and Shoup [12] that provides arbitrarily many unpredictable random bits. It guarantees termination of the agreement protocol within an expected constant number of rounds.

Threshold-cryptographic protocols have been used for secure service replication before, e.g., by Reiter and Birman [56]. However, a major complication for adopting threshold cryptography to our asynchronous distributed system is that many early protocols are not robust and that most protocols rely heavily on synchronous broadcast channels. Only very recently, non-interactive schemes have been developed that satisfy the appropriate notions of security, such as the threshold cryptosystem of Shoup and Gennaro [60] and the threshold signature scheme of Shoup [59]. Both have non-interactive variants that integrate well into our asynchronous model. However, they can be proved secure only in the so-called *random oracle model* that makes an idealizing assumption about cryptographic hash functions [4]. This falls short from a proof in the real world but gives very strong heuristic evidence for their security; there are many practical cryptographic algorithms with proofs only in this model.

2.2 No Timing Assumptions

Like the other parts of this report, this section applies to the specific model used for the design of secure service replication protocols, and has to be seen in the context of the general MAFTIA middleware architecture [64]. Here, we do not make any timing assumptions and work in a completely asynchronous model. (In contrast, the MAFTIA middleware relies on a minimal, trusted time service provided by a specialized subsystem [63].) Asynchronous protocols are attractive because in a synchronous system, one would have to specify timeout values, which is very difficult when protecting against arbitrary failures that may be caused by a malicious attacker.

It is usually much easier for an intruder to block communication with a server than to subvert it. Prudent security engineering also gives the adversary full access to all specifications, including timeouts, and excludes only cryptographic keys from her view. Such an adversary may simply delay the communication to a server for a little longer than the timeout and the server appears faulty to the remaining system.

Time-based failure detectors [16] can easily be fooled into making an unlimited number of wrong failure suspicions about honest parties like this. The problem arises because one crucial assumption underlying the failure detector approach, namely that the communication system is stable for some longer periods when the failure detector is accurate, does not hold against a malicious adversary. A clever adversary may subvert a server and make it appear working properly until the moment at which it deviates from the protocol—but then it may be too late. Heuristic predictions about the future behavior of a server are pointless in security engineering.

Of course, an asynchronous model cannot guarantee a bound on the overall response time of an application. But the asynchronous model can be seen as an elegant way to abstract from time-dependent peculiarities of an environment for proving an algorithm correct such that it satisfies liveness and safety under *all* timing conditions. By making no assumption about time at all, the coverage of the timing assumption appears much bigger, i.e., it has the potential to be justified in a wider range of real-world environments. For our applications, which focus on the security of trusted services, the resulting lack of timeliness seems tolerable.

A variation of the asynchronous model is to assume probabilistic behavior of the communication links [11, 49], where the probability that a link is broken permanently decreases over time. But since this involves a timing assumption, it is essentially a probabilistic synchronous model (perhaps it should also bear that name) and suffers from all the problems mentioned before. The model investigated by Moser and Melliar-Smith [49] assumes, additionally, a fairness property and a partial order imposed by the underlying communication system, but such assumptions seem also difficult to justify on the Internet.

2.3 Static Server Set

Distributing a trusted service among a static set of servers leverages the trust in the availability and integrity of each individual server to the whole system. This set is to remain fixed during the whole lifetime of the system, despite observable corruptions. The reason is that all existing threshold-cryptographic protocols are based on fixed parameters (e.g., n and t) that must be known when the key shares are generated.

A corrupted server cannot be resurrected easily because the intruder may have seen all its cryptographic secrets. Unless specialized "proactive" protocols [13] are used to refresh all key shares periodically, the only way to clean up a server is to redistribute fresh keys. However, proactively secure cryptosystems in asynchronous networks are an open area of research (see Chapter 5).

The alternative is to remove apparently faulty servers from the system. This is the paradigm of view-based group communication systems in the crash-failure model (see the survey in [53]). They offer resilience against crash failures by eliminating non-responding servers from the current view and proceeding without them to the next view. Resurrected servers may join again in later views.

The Rampart toolkit [55] is the only group communication system that uses views and tolerates arbitrary failures. But since it builds on a membership protocol to agree dynamically on the group's composition, it easily falls prey to an attacker that is able to delay honest servers just long enough until corrupted servers hold the majority in the group. Because the maintenance of security and integrity is the primary application of our protocols for trusted services, we cannot tolerate such attacks and use a static group instead (but again, see Chapter 5).

2.4 Related Work

The use of cryptographic methods for maintaining consistent state in a distributed system has a long history and originates with the seminal work of Pease, Shostak, and Lamport [51].

One of the first attempts to build secure and robust replicated services was *DELTA-4*, an EU-funded research project [21]. DELTA-4 developed a general architecture for dependable distributed systems. It provides distributed, intrusion-tolerant services for data storage, authentication and authorization. Secrecy is supported via client-side encryption and data fragmentation, and availability via data replication for the fragments. Secret sharing is supported, but no computations on shared secrets or robust protocols are implemented. DELTA-4 assumes a synchronous communication network, and for the security services a static, threshold-based adversary structure.

The pioneering work of Reiter and Birman [56] (abbreviated RB94 henceforth) introduces secure state machine replication in a Byzantine environment and a broadcast protocol based on threshold cryptography that maintains causality among the requests. Similar to our architecture, it uses a static set of servers, who share the keys of a threshold signature scheme and a threshold cryptosystem. Thus, clients need only know the single public keys of the service, but not those of individual servers.

In order to obtain a fully robust system for an asynchronous environment with malicious faults, however, RB94 must be complemented with robust threshold-cryptographic schemes and secure atomic broadcast protocols, which were not known at that time. Our work builds on this and attempts to close this gap.

Subsequent work by Reiter on *Rampart* [55] shares our focus on distributing trusted services, but assumes a different model as explained in the previous sections: it implements atomic broadcast on top of a group membership protocol that dynamically removes apparently faulty servers from the set.

The broadcast protocols of Malkhi, Merritt, and Rodeh [45] work again with a static group in a model similar to ours, but implement only reliable broadcast and do not guarantee a total order, as needed for maintaining consistent state.

The *e-Vault* prototype for secure distributed storage [30] addresses a subset of the applications considered here, namely storing and retrieving immutable data. It works in a synchronous environment, though, and is not directly applicable to wide-area networks.

Castro and Liskov [15] (called CL99 below) present an interesting practical algorithm for distributed service replication that is very fast if no failures occur. It requires no explicit timeout values, but assumes that message transmission delays do not grow faster than some predetermined function for an indefinite duration. Since the CL99 protocol is deterministic, it can be blocked by a Byzantine adversary (i.e., violating liveness), but it will maintain safety under all circumstances. In contrast, our approach satisfies both conditions because it is based on a probabilistic agreement protocol.

The *Fleet* architecture of Malkhi and Reiter [47] supports loose coordination in large-scale distributed systems and shares some properties of our model. It works in a Byzantine environment and uses quorum systems and threshold cryptography for implementing a randomized agreement protocol (in the form of "consensus objects"). However, the servers do not directly communicate with each other for maintaining distributed state and merely help clients carrying out fault-tolerant protocols. Close coordination of all servers is also not a primary goal of Fleet. Implementing distributed state machine replication on top of Fleet is possible, in principle, but needs additional steps.

The *Total* family of algorithms for total ordering by Moser and Melliar-Smith [49] implements atomic broadcast in a Byzantine environment, but only assuming a benign network scheduler with some specific probabilistic fairness guarantees. Although this may be realistic in highly connected environments with separate physical connections between all machines, it seems not appropriate for arbitrary Internet settings.

SecureRing [39] and the very recent work of Doudou, Garbinato, and Guerraoui [24] (abbreviated as DGG00) are two examples of atomic broadcast protocols that rely on failure detectors in the Byzantine model. They encapsulate all time-dependent aspects and obvious misbehavior of a party in the abstract notion of a failure detector and permit clean, deterministic protocols (see also [3]). However, most implementations of failure detectors will use timeouts and actually suffer from some of the problems mentioned above. It also seems that Byzantine failure detectors are not yet well enough understood to allow for precise definitions.

A tabular comparison of systems for secure state machine replication is shown in Table 2.1. We think the cryptographic model with randomized Byzantine agreement is both practically and theoretically attractive, although it seems to have been somewhat overlooked in the past. (The fact that randomized agreement protocols may not terminate with non-zero probability does not matter because this probability is negligible; moreover, if a protocol involves *any* cryptography at all, and the practical protocols mentioned above do so, a negligible probability of failure cannot be ruled out.) Remarkably, during the two decades since the question of maintaining "interactive consistency" was first formulated [51], no secure system in our asynchronous model has been designed until very recently.

Reference	Timing	Servers	BA?	Remark
RB94 [56]	async.	static	yes$^{(1)}$	crash-failures only
Rampart [55]	async.	dynamic	no	FD for liveness and safety
Total alg. [49]	prob. async.	static	no	needs causal order on links
CL99 [15]	async.	static	no	FD for liveness
Fleet [47]	async.	static	yes$^{(2)}$	no state machine replication
SecureRing [39]	async.	static	yes$^{(3)}$	"Byzantine" FD
DGG00 [24]	async.	static	yes$^{(3)}$	"Byzantine" FD
this work	async.	static	yes$^{(4)}$	general adversaries (Q^3)

Table 2.1: Systems for secure state machine replication (Fleet supports only loose coordination, but not state machine replication directly). All systems achieve optimal resilience $t < n/3$. The column entitled "BA?" notes if a system solves Byzantine agreement (BA); those who do build (1) on an (assumed) atomic broadcast protocol, (2) on randomization and threshold signatures, (3) on a failure detector or "muteness detector" in the Byzantine model, or (4) on a cryptographic coin [12] in the underlying Byzantine agreement protocol. Some systems need a failure detector (FD); all except Total need a trusted dealer for setup. Fleet can also tolerate adversaries of Byzantine Quorum systems, our system tolerates general Q^3-adversaries.

2.5 Generalized Adversary Structures

The common approach in fault-tolerant distributed systems is that at most a fraction of all servers fail. This model is based on the assumption that faults occur independently of each other and affect all servers equally likely. For random and uncorrelated faults within a system as well as isolated external events this seems adequate.

However, faults that represent malicious acts of an adversary may not always match these assumptions. This causes a conceptual obstacle for using the replication-based approach to achieve security in adversarial environments. In our setting, for example, if all servers in the system have a common vulnerability that permits a successful attack by an intruder, the integrity of the whole system may be violated easily. The independence assumption applies here only to the extent that the work needed for breaking into a server is the same for each machine. With the sophisticated tools, automated exploit scripts, and large-scale coordinated attacks found on the Internet today, this assumption becomes increasingly difficult to justify.

2.5.1 Concept

One solution for this problem, which we propose here, is to use *generalized adversary structures*. They can accommodate a strictly more general class of failures than with any weighted threshold structure. In the Byzantine model, a collection of corruptible servers is also called an *adversary structure*. Such an adversary structure specifies the subsets of parties that may be corrupted at the same time.

We describe two concrete instantiations of such general adversary structures that are based on a classification of all servers according to one or more attributes with at least four values each.

Generalized adversary structures for secure fault-tolerant computing are also used in Byzantine Quorum systems [46] and the synchronous Byzantine agreement protocol of Fitzi and Maurer [28]; for combining them with threshold cryptography, we are restricted to those that correspond to linear secret sharing schemes (based on the results of Cramer, Damgård, and Maurer [19]).

Let $\mathcal{P} = \{1, \ldots, n\}$ denote the index set of all parties $P_1, \ldots, P_n$. The *adversary structure* $\mathcal{A}$ is a family of subsets of $\mathcal{P}$ that specifies which parties the adversary may corrupt. $\mathcal{A}$ is monotone (i.e., $S \in \mathcal{A}$ and $T \subset S$ imply $T \in \mathcal{A}$) and uniquely determined by the corresponding maximal adversary structure $\mathcal{A}^*$ in which no subset contains another one. For the traditional threshold model of at most a certain number of corrupted parties, the adversary may corrupt up to t arbitrary parties. In this case, $\mathcal{A}^*$ contains all subsets of $\mathcal{P}$ with cardinality t.

Most protocols impose certain restrictions on the type of corruptions that they can tolerate. For a threshold adversary in an asynchronous distributed system model, $n > 3t$ is in general a necessary and sufficient condition. The analogous condition for protocols with a general adversary structure $\mathcal{A}$ is the so-called Q^3 *condition* [38]: no three of the sets in $\mathcal{A}$ cover $\mathcal{P}$. (Note that $n > 3t$ is a special case of this.)

The adversary structure specifies the (maximally) corruptible subsets of parties. Its complement is called the *access structure* and specifies the (minimally) qualified subsets that are needed to take some action. For example, it is used in secret sharing in cryptography [62], where it denotes the sets of parties who may reconstruct the shared secret. The access structure is usually the more important tool for the protocol designer than the adversary structure. In the example of the threshold system above, all sets of $t+1$ or more parties belong to the access structure.

2.5.2 Designing Protocols for General Adversary Structures

Every adversary structure can also be described by a Boolean function g on n variables that represent a subset of $\mathcal{P}$ as follows. We associate a subset of $\mathcal{P}$ with its *characteristic vector* whose ith element is 1 if and only if P_i is in the subset. Extending the domain of g to all subsets of $\mathcal{P}$ in this way, g outputs 0 on all elements of the adversary structure (i.e., for all sets of parties that might be corrupted by the adversary), and 1 otherwise. To represent g, we use arbitrary fan-in threshold gates Θ_k^n that output 1 if and only if at least k of their inputs are 1 (note that AND and OR gates correspond to the special cases Θ_n^n and Θ_1^n). For example, $g(S) = \Theta_{t+1}^n(S)$ in the threshold example.

Although they are not described in this form, all threshold-cryptographic protocols used by our architecture (cf. Chapter 3) can be extended to a generalized Q^3 adversary structure $\mathcal{A}$ for which the corresponding secret sharing access structure can be implemented by a linear secret sharing scheme. This requires changing some details in the cryptographic operations, but there are no essential difficulties. The agreement and broadcast protocols need to be changed as follows:

- Where a set of $n - t$ values is required, take all values in $\mathcal{P} \setminus S$ for some $S \in \mathcal{A}^*$.
- Where $2t + 1$ values are needed, take all values in $S \cup T \cup \{i\}$ for any $S, T \in \mathcal{A}^*$ with $S \cap T = \emptyset$ and $i \notin S \cup T$.
- Where $t + 1$ values are needed, take all values in $S \cup \{i\}$ for any $S \in \mathcal{A}^*$ and $i \notin S$.

2.5.3 Differentiating Servers by Attributes

Suppose there is an attribute of all parties in the system that takes on at least four different values. If the characteristics of corrupting a party vary with the attribute, then this classification can be exploited directly to design a system in which all parties in the same class may be corrupted simultaneously. With $n = 4$ this reduces to the threshold case. For example, the servers in a wide-area distributed system may vary by physical location, logical domain, system management personnel, type of operating system, other applications running on the same machine, and implementation of the protocols. All of these are suitable attributes.

We do not make any further distinctions between the attribute values here; this leads naturally to a threshold failure model in the attribute dimension. However, arbitrary and complex relations between the parties can be modeled as long as there exists a corresponding linear secret sharing scheme. For simple linear relations, traditional weighted thresholds may already be enough, which can be obtained by grouping together several logical parties to a single physical party.

We give two examples of generalized adversary structures by describing the linear secret sharing schemes on which they are based. The first one shows how to combine attribute classification with the traditional threshold model, the second one is based on combining two separate classifications.

Before we continue, we need to introduce some notation. Let *class* denote the name of an attribute and also the set of attribute values. For $c \in \mathit{class}$, define $\chi_c : 2^{\mathcal{P}} \to \{0,1\}$ as the characteristic function of the attribute on a set of parties such that $\chi_c(S) = 1$ if and only if $\mathit{class}(i) = c$ for some $i \in S$.

Example 1. Consider a system of nine servers and one attribute $\mathit{class} = \{a, b, c, d\}$ that satisfies

$$\begin{array}{rcrcl} \mathit{class}(1) & = \cdots = & \mathit{class}(4) & = & a \\ \mathit{class}(5) & = & \mathit{class}(6) & = & b \\ \mathit{class}(7) & = & \mathit{class}(8) & = & c \\ & & \mathit{class}(9) & = & d. \end{array}$$

This could be the operating system of a server, for example, with class a representing a common but not very secure operating system and class d representing the most secure one of the four *class* values.

We want to design a system that tolerates the corruption of at most two arbitrary servers or all servers in any particular class; its adversary structure $\mathcal{A}_1$ is given by

$$g(S) \;=\; \overline{\Theta_3^9(S)} \vee \overline{\Theta_2^4(\chi_a(S), \chi_b(S), \chi_c(S), \chi_d(S))}.$$

$\mathcal{A}_1^*$ consists of $\{1, \ldots, 4\}$ and of all pairs of servers that are not both of class a. The corresponding access structure for secret sharing is given by

$$\overline{g(S)} \;=\; \Theta_3^9(S) \wedge \Theta_2^4(\chi_a(S), \chi_b(S), \chi_c(S), \chi_d(S)).$$

In other words, secrets may be reconstructed by coalitions of servers of size at least three that also cover at least two different classes.

One may readily verify that $\mathcal{A}_1$ satisfies the Q^3 condition. The corresponding linear secret sharing scheme follows directly from the expression for $\overline{g(S)}$ using the standard construction of Benaloh and Leichter [8]. The agreement and broadcast protocols can be adapted according to the modifications sketched above.

Example 2. The classification method works simultaneously for an arbitrary number of attributes and attribute values. We illustrate this for two attributes with four values each, denoted by $class_1 = \{a, b, c, d\}$ and $class_2 = \{\alpha, \beta, \gamma, \delta\}$. Assume all combinations of $class_1$ and $class_2$ exist so that $\mathcal{P}$ contains at least sixteen servers.

To be concrete, think of a distributed system of sixteen servers implementing a secure directory service for a large company that is running on nodes in New York (USA), Tokyo (Japan), Zurich (Switzerland), and Haifa (Israel) and consists of servers running AIX, Windows NT, Linux, and Solaris operating systems. Thus, $class_1$ corresponds to the location and $class_2$ to the operating system of a server.

We can now obtain a distributed system that, for example, tolerates the *simultaneous* corruption of all servers with a particular operating system and all servers in one location. Its adversary structure is characterized by

$$g(S) \;=\; \overline{\Theta_2^4(x_a, x_b, x_c, x_d)} \vee \overline{\Theta_2^4(y_\alpha, y_\beta, y_\gamma, y_\delta)},$$

where for $v \in class_1$,

$$x_v = \Theta_2^4(\chi_v(S) \wedge \chi_\alpha(S),\ \chi_v(S) \wedge \chi_\beta(S),\ \chi_v(S) \wedge \chi_\gamma(S),\ \chi_v(S) \wedge \chi_\delta(S))$$

and for $\nu \in class_2$,

$$y_\nu = \Theta_2^4(\chi_a(S) \wedge \chi_\nu(S),\ \chi_b(S) \wedge \chi_\nu(S),\ \chi_c(S) \wedge \chi_\nu(S),\ \chi_d(S) \wedge \chi_\nu(S))$$

This adversary structure satisfies the Q^3 condition, as can be verified easily. The corresponding secret sharing scheme is characterized by the negated expression, $g(S) = \Theta_2^4(x_a, x_b, x_c, x_d) \wedge \Theta_2^4(y_\alpha, y_\beta, y_\gamma, y_\delta)$. Intuitively, the sharing introduces two secret values (one for each class) at the top level that must both be known to recover the secret. To reconstruct the first one, at least two of the four $class_1$ points $x_a, \ldots, x_d$ must be known; each point in turn can again only be reconstructed by a subset that covers at least two $class_2$ values *and* the corresponding $class_1$ value. In other words, x_a is shared among the four parties with a $class_1$ value of a using a two-out-of-four scheme, etc. The top-level secret for $class_2$ is distributed analogously.

The resulting distributed system maintains liveness and safety as long as there are servers with three operating systems at three locations that are uncorrupted; but one location may be unreachable and one operating system could contain easily exploitable vulnerabilities so that a maximum of seven servers could have failed at any moment. Note that all solutions based on thresholds can tolerate at most five corruptions among the 16 servers.

2.6 System Initialization

Our distributed system architecture uses several broadcast protocols which involve threshold cryptography primitives. In order to engage in such threshold-cryptographic protocols, the servers need specific cryptographic keys, which are established during the initialization phase. More precisely, in the initialization phase we run several key generating algorithms on the inputs k, n, t, and κ, where k is a suitably chosen security parameter, n is the number of participants, t is the number of corrupted parties, and κ is the combining threshold such that $t < \kappa \leq n - t$ (usually $\kappa = t + 1$). At the end of the initialization phase, each party (or server) must have all its secret key shares in its private storage.

The above description assumes that a trusted dealer creates all those keys; however, this introduces a single point of failure and it would be desirable to avoid this. The alternative is to use a distributed key generation protocol by all parties in the system.

In this section we describe all cryptographic keys that have to be provided during initialization and discuss their generation by a trusted dealer and by distributed protocols.

2.6.1 Types of Cryptographic Keys

According to the previous sections, we need keys for the following cryptographic schemes:

Threshold signature scheme: We use the non-interactive threshold signature scheme by Shoup [59], which is based on RSA [57] and can be proven secure in the random oracle model assuming the RSA problem is hard. Its key-generation algorithm must provide the public key of the signature scheme PK, a global verification key VK, n secret key shares, $SK_1, \ldots, SK_n$, and n local verification keys $VK_1, \ldots, VK_n$. The initial state information for server i consists of the secret key SK_i along with all verification keys.

Threshold cryptosystem: We use the non-interactive threshold cryptosystem that is secure against chosen-ciphertext attacks proposed by Gennaro and Shoup [60]. It is proven secure in the random oracle model assuming the hardness of the Diffie-Hellman problem. The keys for this scheme are discrete logarithm-based Diffie-Hellman public keys. The key generation algorithm outputs a public encryption key E, and a list of secret decryption key shares $D_1, \ldots, D_n$. The initial state information for the threshold encryption scheme for server i consists of the decryption key share D_i along with the public encryption key E.

Threshold coin-tossing: Cachin et al. [12] mention two implementations of a threshold coin-tossing scheme: based on deterministic threshold signatures or a novel protocol based on discrete logarithms. The first implementation simply uses the hash of a threshold signature on the name of a coin as the coin value. It can be based on the RSA-threshold signature scheme by Shoup [59] mentioned above. The second implementation uses exactly the same keys as the discrete-logarithm-based threshold cryptosystem mentioned above [60].

Digital signature scheme: Our architecture assumes that every party can issue digital signatures under its signing key, for example, RSA signatures [57]. In order to obtain the necessary keys, a public signature verification key and a private signing key must be produced for each party. After initialization, each server must know its private signing key and all public verification keys.

Message authentication: For message authentication, a symmetric secret key for every pair of servers is needed (using any MAC algorithm [48]). Generating these keys is straightforward. After initialization, each server must know those symmetric keys that it shares with all other servers.

Since the keys for coin-tossing are the same as for either threshold signatures or threshold public-key encryption, we do not mention them further. In conclusion, the initial state information of each server contains: one secret key and all verification keys of the threshold signature scheme, one public encryption key and one secret decryption key of the threshold cryptosystem, one private signing key and all public keys for the digital signature scheme, and $n-1$ symmetric keys for message authentication.

2.6.2 Key Generation By Trusted Dealer

Using a trusted dealer, key generation works as follows. A special party, the dealer, runs the algorithms for generating all necessary keys and outputs n files, each containing the initial state information for one party. This key file is then given to the corresponding party, i.e., included in its initial state, before the party starts operating. There exist several methods for distributing these files to the servers. If an authenticated public key for encryption is known of each party, the dealer can encrypt the information and send it over a public network. Otherwise, external authentication must be used, and the file could be distributed on a floppy disk by a trusted courier or sent by registered mail.

Generating and distributing these keys must occur securely and be protected from the adversary. The dealer must be trusted to perform the operations correctly and not to leak any information about the keys. It is best to assume that all records and information pertaining to the key generation are destroyed afterwards.

RSA-based threshold signature keys: The keys of the threshold signature schemes are the public key PK, consisting of a standard RSA public key pair (N, e), where N is the product of two safe primes and e a prime greater than n (the number of participating servers). The algorithm for creating the RSA public key pair (N, e) does the following:

- Select two large prime numbers p' and q', compute $p = 2p' + 1$ and $q = 2q' + 1$, and test whether p and q are prime, and repeat until the test outputs "yes" for both. (This can be done with a Rabin-Miller primality test.)
- Let $N = pq$ and $\phi(N) = (p - 1)(q - 1)$.
- Choose a number d that does not divide $\phi(N)$
- Let e be such that $ed \equiv 1 \mod \phi(N)$.

The number d is input to the generation algorithm for the secret key shares. The algorithm selects a random polynomial $f(x) \in \mathbb{Z}_{odd(\phi(N))}[x]$ of degree t with its constant term set to d. Each secret key SK_i (for $i = 1, \ldots, n$) results from evaluating f on i, i.e., $SK_i = f(i)$ computed in $\mathbb{Z}_{odd(\phi(N))}$. (Note that $odd(\phi(N)) = p'q'$ is the product of all odd prime factors of $\phi(N)$.)

Then the global verification key VK is chosen randomly in the subgroup of all squares in $\mathbb{Z}_N^*$, and the verification keys VK_i are generated by computing $VK^{f(i)}$ in $\mathbb{Z}_N^*$ for $i = 1, \ldots, n$.

Discrete-logarithm-based keys: For the threshold public-key cryptosystem, a shared Diffie-Hellman public key is generated as follows. Choose randomly a large prime q and a generator g of a group G such that G is of order q. Choose a random polynomial $f(x) \in \mathbb{Z}_q[x]$, let $x_i = f(i)$ for $i = 0, \ldots, n$, and let $y = g^{x_0}$. The key shares are $D_i = (y, h_i, \ldots, h_n)$, where $h_i = g^{x_i}$ for $i = 1, \ldots, n$, and the public key is $E = y$.

Signature keys and authentication keys: Generating these keys by a dealer is straightforward [48].

2.6.3 Key Generation by Distributed Protocols

Unfortunately, assuming a trusted dealer for key-distribution introduces a single point of attack. One can avoid this by using a distributed protocol for key generation among the servers that (ideally) tolerates the same type of faults as the distributed system. Consequently, this has to be done without revealing the private keys in the initial state information, which should not be known by the other servers.

However, no efficient key generation algorithms are currently known in our asynchronous cryptographic system model. There are protocols that work in a synchronous setting and assume that all servers are connected by an authenticated broadcast channel. Since the initialization process depends on a synchronized external action anyway, it seems reasonable to exploit synchrony also for a distributed system initialization protocol.

It should be noted that any distributed protocol will need to authenticate the participating servers at the very least. Thus, key generation for the signature scheme (or a public-key infrastructure) cannot be realized fully by a distributed, fault-tolerant protocol.

Key generation protocols for discrete logarithm-type public keys are well known. See, for example, the work of Gennaro et al. [32] and the references therein.

For generating ordinary RSA threshold public keys, Boneh and Franklin [9] recently introduced a scheme in which three (of more) parties are able to generate a shared RSA key without a trusted dealer. However, they proved their scheme secure only in the honest-but-curious case. Later Frankel, MacKenzie, Yung [29] added some techniques to the original scheme to achieve robustness, i.e., the shared key generation is possible even in the presence of malicious parties.

It is currently not known, however, how to efficiently generate RSA public keys of the special form needed by the threshold signature scheme mentioned above (i.e., N being the product of two safe primes). Below we sketch how such a protocol for distributively generating the product of two safe primes might work.

The generated RSA modulus has to be the product of two safe primes, i.e., $N = pq$, where $p = 2p' + 1$ and $q = 2q' + 1$ are prime and p' and q' are itself prime numbers. On a high level, such a protocol proceeds as follows:

1. Each server picks an integer p_i and keeps it secret. Using a private distributed computation, the servers make sure that $p = \sum_{i=1}^{n} p_i$ is not divisible by any prime less than some bound B. If this step fails, they repeat it until the condition holds. $q = \sum_{i=1}^{n} q_i$ is computed analogously.

2. Using a private distributed computation, the servers compute $N = \left(\sum_{i=1}^{n} p_i\right) \cdot \left(\sum_{i=1}^{n} q_i\right)$. Having obtained N as the proposed public key, the parties perform a trial division to test if N is not divisible by small primes in a certain range. Then the servers engage in a private distributed computation to test that N is indeed the product of two primes. If the test fails, the protocol is restarted from the first step.

3. Finally the servers engage in a second private distributed computation to test whether the two primes the modulus exists of are safe primes or not. If the test fails, the protocol is restarted from the first step.

4. They output N as the modulus, and perform another distributed computation to obtain a public exponent e and, from p_i and q_i, the corresponding secret key shares SK_i, the verification key VK, and the local verification keys VK_i for $i = 1, \ldots, n$.

2.7 Relation to MAFTIA Middleware Architecture

The architecture described in this report is a special case of the general service and protocol architecture of the MAFTIA middleware, as described in the companion report [64]. This general architecture aims at integrating all concepts that are relevant for typical distributed Internet-based applications, and is therefore conceptionally much richer than the small subset needed here.

The general system model supports different models for faults, different levels of synchronization among parties, a general hierachical network topology, miscellaneous interaction styles among the parties, and different types of groups (see Section 2 of [64]). Here we use one specific incarnation of this system:

- We use a *controlled failure assumption,* but controlled in a very weak sense only: we just assume that the adversary cannot break our cryptographic primitives. (Chapter 5 sketches an extension to a *hybrid failure* assumption.)
- We follow the *time-free approach*, assuming an *asynchronous network* (see Section 2.2).

  The general model of [64] defines a hybrid model of synchrony, the *Trusted Timely Computing Base* (TTCB). The TTCB defines a core part of a system which is essentially synchronous and thus provides timeliness. The other parts of a system might be arbitrarily asynchronous (see [64], Section 2.2 for details), and are used for all those interaction where timeliness is not essential, or can be achieved indirectly (by using the TTCB as a means for recovery from timing failures). Thus, the TTCB allows to combine the best of both worlds.

  The applications considered here have no strict real-time requirements, and thus timeliness is a less important issue. On the other hand, our applications should be secure even against the strongest adversaries, e.g., one that can influence the synchronization among parties. Thus, the time-free approach offers a sufficient quality of service, and allows us to be on the safe side. We may envision using security-related services provided by the TTCB, if that improves the efficiency and complexity of the protocols.
- In principle, our applications could run on any network topology, but from a security point of view we require a high degree of connectivity and, for servers, independence of sites and participants (in terms of [64]). The latter is given by the *generalized*

adversary structures above, which describe the sets of servers that might fail simultaneously.

- All our applications are instances of a *client-server* model with replicated servers (in the sense described earlier). The group of servers uses a *multipeer* interaction style.

  Viewed as an instance of the MAFTIA middleware, our architecture supports a *client-server* model for *transactional services.*

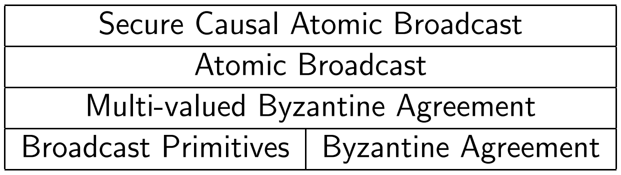
- Our applications assume an *open, dynamic group* of remote clients and a *closed, static group* of servers. See Section 2.3 for more details.

The general service architecture of [64], Section 3, provides a structure for the components of a MAFTIA node and identifies the services that shall be provided by the MAFTIA middleware. The specific architecture described here provides a subset of these services, as needed for our applications.

More specifically, at the Site Level we assume a Multipoint Network as described in [64], Section 3. On top of this network we implement time-free versions of all Communication Support services except those related to time and clock synchronization (which are not needed here). On the Participant Level we support several Activity Services–such as static replication management, key management and transactional management–always assuming the special system model sketched above and described earlier in this section.

3 Architecture

This chapter presents a protocol architecture for multiple broadcast protocols and implementations of broadcast protocols, as used by our distributed trusted services. We need protocols for basic, reliable broadcast, atomic broadcast, and secure causal atomic broadcast (see below); they can be described and implemented in a modular way as follows, using multi-valued Byzantine agreement and randomized binary Byzantine agreement primitives.


Secure Causal Atomic Broadcast
Atomic Broadcast
Multi-valued Byzantine Agreement
Broadcast Primitives
Byzantine Agreement

All our broadcast and agreement protocols work under the optimal assumption that $n > 3t$.

Byzantine agreement requires all parties to agree on a binary value that was proposed by an honest party. The protocol of Cachin et al. [12] follows the basic structure of all randomized solutions (e.g., [5]) and terminates within an expected constant number of asynchronous rounds. It achieves the optimal resilience $n > 3t$ by using a robust threshold coin-tossing protocol, whose security is based on the so-called Diffie-Hellman problem. It requires a trusted dealer for setup, but can process an arbitrary number of independent agreements afterwards. Threshold signatures are further employed to decrease all messages to a constant size. As mentioned before, its security proof relies on the random oracle model.

A useful primitive is also *multi-valued Byzantine agreement*, which provides agreement on values from larger domains. Multi-valued agreement requires a non-trivial extension of the binary Byzantine agreement protocols mentioned above. The difficulty in multi-valued Byzantine agreement is how to ensure the "validity" of the resulting value, which may come from a domain that has no a priori fixed size. One cannot tolerate the agreement protocol to decide on a value that no party proposed. Our implementation of multi-valued Byzantine agreement uses only a constant expected number of rounds.

A basic broadcast protocol in a distributed system with failures is *reliable broadcast*, which provides a way for a party to send a message to all other parties. Its specification requires that all honest parties deliver the same set of messages and that this set includes all messages broadcast by honest parties. However, it makes no assumptions if the sender of a message is corrupted and does not guarantee anything about the order in which messages are delivered. The reliable broadcast protocol of our architecture is an optimized variant of the elegant protocol by Bracha and Toueg [11]. We also use a variation of it, called *consistent broadcast*, which is advantageous in certain situations. It guarantees uniqueness of the delivered message (thus the name consistent broadcast), but relaxes the requirement that all honest parties actually deliver the message—a party may still learn about the existence of the message by other means and ask for it. A similar protocol has also been used by Reiter [54].

An *atomic broadcast* guarantees a total order on messages such that honest parties deliver all messages in the same order. Any implementation of it must implicitly reach agreement whether or not to deliver a message sent by a corrupted party and, intuitively, this is where Byzantine agreement is needed. The basic structure of the atomic broadcast protocol follows the atomic broadcast protocol of Chandra and Toueg [16] for the crash-failure model: the parties proceed in global rounds and agree on a set of messages to deliver at the end of each round.

Multi-valued Byzantine agreement is used to determine the set of messages to be delivered in the current round. All selected messages are then delivered according to a fixed order. This atomic broadcast protocol guarantees liveness, i.e., a message broadcast by an honest party cannot be delayed arbitrarily by the adversary once it is known to at least $t + 1$ parties.

A *secure causal atomic broadcast* is an atomic broadcast that also ensures a causal order among all broadcast messages, as put forward by Reiter and Birman [56]. It can be implemented by combining an atomic broadcast protocol that tolerates a Byzantine adversary with a robust threshold cryptosystem. Encryption ensures that messages remain secret up to the moment at which they are guaranteed to be delivered. Thus, client requests to a trusted service using this broadcast remain confidential until they are scheduled and answered by the service. The threshold cryptosystem must be secure against adaptive chosen-ciphertext attacks to prevent the adversary from submitting any related message for delivery, which would violate causality in our context. Maintaining causality is crucial in the asynchronous environment for replicating services that involve confidential data.

The protocol for secure causal atomic broadcast follows the basic idea of Reiter and Birman's protocol. By using the robust atomic broadcast mentioned before and the recent, non-interactive threshold cryptosystem [60], it is actually the first secure implementation that we are aware of, and it can be proved secure in the random oracle model.

3.1 Formal Model

This section describes a formal model for our modular protocol architecture, where a number of parties communicate over an insecure, asynchronous network, and where an adversary may corrupt some of them.

Our model differs in two respects from other models traditionally used in distributed systems with Byzantine faults:

1. In order to use the proof techniques of complexity-based cryptography [33], our model is *computational*: all parties and the adversary are constrained to perform only feasible, i.e., polynomial-time, computations. This is necessary for using formal notions from cryptography in a meaningful way.
2. We make no assumptions about the network at all and leave it under complete control of the adversary. Our protocols work only to the extent that the adversary delivers messages faithfully. In short, *the network is the adversary.*

The differences become most apparent in the treatment of termination, for which we use more concrete conditions that together imply the traditional notion of "eventual" termination.

We define termination by bounding a *statistic* measuring the amount of work that honest, uncorrupted parties do on behalf of a protocol; in particular, we use the communication complexity of a protocol for this purpose. Since the specification of a protocol requires certain things to happen under the condition that all protocol messages have been delivered, bounding the length (and also the number) of protocol messages generated by uncorrupted parties ensures that the protocol has actually terminated under this condition. In cryptography one proves security with respect to all polynomial-time adversaries, and we adopt this model here as well. Our notion of an *efficient* (deterministic) protocol requires that the statistic is bounded by a fixed polynomial, which is independent of the adversary. As we rely on randomization (for Byzantine agreement as well as for other things), we also define a corresponding probabilistic bound for randomized protocols; from this a bound on the expected running time of a protocol can be derived. Both of our notions are closed under modular composition of protocols, which is not trivial for randomized protocols.

Among the many established formal models for asynchronous distributed protocols, the I/O automata model of Lynch and Tuttle [42, 44, 43] seems to be the most general one. It has also been extended to allow for modeling of randomized protocols. But even though authentication and digital signatures have been used before in secure distributed protocols, apparently no adequate formal model has integrated both approaches before [43, p. 115].

3.1.1 Basic System Model

The security parameter of our computational security model is denoted by k. A quantity ϵ is called *negligible* (as a function of k) if for all $c > 0$ there exists a k_0 such that $\epsilon(k) < \frac{1}{k^c}$ for all $k > k_0$. As k is usually not mentioned explicitly, keep in mind that all parameters are bounded by polynomials in k.

3.1.1.1 Parties and Protocols

Multi-Party Protocols. An n-party protocol consists of a collection of n parties, $P_1, \ldots, P_n$, which are probabilistic interactive Turing machines that run in polynomial time (in k). Such a machine has two dedicated interfaces for reading incoming messages and writing outgoing messages. There is also an initialization algorithm, which is run by an additional party called the dealer; on input k, n, and t, it generates the state information that is used to initialize each party. For simplicity, assume $n \leq k$.

After initialization, a party P_i may be activated repeatedly with some input message. It will carry out some computation, update its state, possibly generate some output messages, and wait for the next activation.

We leave it to the adversary to choose n and t, but a specific protocol might impose its own restrictions (e.g., $t < n/3$). We can assume that the dealer includes these values, as well as the index i, in the initial state of P_i.

Our model includes a public-key infrastructure for digital signatures, i.e., the dealer generates a key pair for a digital signature scheme $\mathcal{S}$ for each party, and includes in the initial state of each party its private key and the public keys of all parties. The dealer initializes a fixed number of threshold cryptosystems as required by the implemented protocols.

The dealer may also generate a public output for information associated with the n-party protocol; this information may be useful for clients of a replicated service that is implemented by the n-party protocol.

Executions and the Adversary. As our network is insecure and asynchronous, protocol execution is defined entirely via the adversary. The adversary is a polynomial-time interactive Turing machine that schedules and delivers all messages and corrupts some parties.

After the initial setup phase, the adversary repeatedly activates a party with some input message(s) and waits for the party to generate some output message(s). The output is given to the adversary and perhaps indicates to whom these messages should be sent, and the adversary may choose to deliver these messages faithfully at some time. But in general, the adversary chooses to deliver any message it wants, or no message at all; we sometimes impose additional restrictions on the adversary's behavior, however.

The adversary also *corrupts* t parties. W.l.o.g. any adversary that corrupts fewer than t parties can be converted into one that corrupts exactly t parties. This simplification seems justified for distributed systems with Byzantine faults where one cannot rely on the actions of a single, potentially corrupted party; all our intended applications are be based on the behavior of (a majority of) the uncorrupted parties.

One distinguishes between *static* and *adaptive* corruptions in cryptography: in the *static* corruption model, the adversary must decide whom to corrupt independently of the execution of the system, whereas in the *adaptive* corruption model, the adversary can adaptively choose whom to corrupt as the attack is ongoing, based on information it has accumulated so far. We adopt a *static* adversary in this work for using the threshold coin-tossing scheme and the Byzantine agreement protocol of Cachin, Kursawe, and Shoup [12], the threshold cryptosystem of Shoup and Gennaro [60], and the threshold signature scheme of Shoup [59]. All of these assume static corruptions.

The adversary receives the initial state of the corrupted parties as produced by the dealer. Otherwise, the corrupted parties are simply absorbed into the adversary: we do not regard them as system components. Uncorrupted parties are called *honest*.

Our formal model leaves control over the application interface for invoking broadcasts and starting agreement protocols up to the adversary. The protocol definitions merely state that *if* the adversary invokes the protocol in a certain way—in the same way an intended application would do—*then* the protocol should satisfy some specific conditions. This reflects that applications might be partially influenced by an adversary, which might cause some security problems if this is not allowed. For simplicity, the application program interface is also mapped onto the single messaging interface, described next.

Modular Protocol Architecture. We describe a modular protocol architecture, in which multiple broadcasts and transactions may execute in parallel. These protocol instances run concurrently and may also invoke other protocol instances on their behalf as sub-protocols. The dynamic relation between all concurrently running protocol instances is given by a directed acyclic graph in which every sub-protocol points to its parent. The "root" protocols with no parents represent instances directly invoked by a user application; in our formal model, they are invoked by the adversary. All other protocol instances are invoked as sub-protocols of some already running parent instance. The reason for letting the adversary invoke the root protocol instances is that a protocol that meets our specifications will behave accordingly in any application, since our definition requires that it works for *any* adversary. Applications whose behavior can be influenced by an adversary are common in distributed systems with security demands.

To identify protocol instances, we assume that each instance is associated with a unique *tag ID*. The value ID is an arbitrary bit string whose structure and meaning are determined by a particular protocol and application; in our formal model, the tag of the root instances is chosen by the adversary because the adversary invokes them. Sub-protocols are identified by hierarchical tags of the form $ID|ID'|\ldots$. The tag value $ID|ID'$ typically identifies a sub-protocol of the parent protocol instance ID and is determined by the parent. The adversary may not introduce a new tag on its own if this extends any previously introduced tag, i.e., the set of tags chosen directly by the adversary must be prefix-free.

3.1.1.2 Communication

Messages. The protocols are described in terms of a single communication interface to which the adversary delivers messages. Each party runs an internal *scheduler* that delivers messages to the protocol instance associated with the corresponding ID. The message interface is used in two different ways, however: to send and receive messages via the network and as a placeholder for local invocation of sub-protocols. Syntactically, invoking a sub-protocol appears as if it were a request of the adversary in our formal model (as mentioned above). Since our protocol specifications guarantee certain behavior when requests come from an arbitrary adversary, an application using a sub-protocol can benefit from this universality, as long as it meets the requirements in the respective specification. The detailed mechanism for composing protocols is part of the scheduler described below.

There are three different types of messages: input actions, output actions, and protocol (implementation) messages. *Input* and *output* actions represent local events and provide local input or carry local output to or from a protocol instance, which might be a sub-protocol of an already running protocol. On the "protocol stack" of the layered architecture, input and output actions travel vertically: inputs "down" to sub-protocols and outputs "up" to higher-layer protocols. All other messages are *protocol messages*, generated and processed by the protocol implementation; they are intended for the peer instances running at other parties on the same level of the stack (directed "horizontally").

These messages are internal implementation messages and they are distinct from the messages actually disseminated as payloads of the broadcast protocols; such messages are sometimes explicitly called *payload* messages.

An *input action* is a message of the form

$$(\mathit{ID}, \texttt{in}, \mathit{action}, \dots),$$

where *action* is specific to the protocol and followed by arbitrary data. Input actions represent local invocations of a protocol, either as a root protocol instance by the adversary or as a sub-protocol of an already running protocol instance. An input action is used to request a service from the protocol instance. There is a special input action *open*, represented by

$$(\mathit{ID}, \texttt{in}, \texttt{open}, \mathit{type}),$$

which must precede any other input action with tag *ID*. When P_i processes such a message with tag *ID* for the first time, it initializes the instance; *type* specifies the type of the protocol being initialized. We say that P_i has *opened* a "channel" with tag *ID* or *activated* a "transaction" with tag *ID*. (Although it is a crucial element, it occurs mostly implicitly before the first regular input action.)

An *output action* is a message of the form

$$(\mathit{ID}, \texttt{out}, \mathit{action}, \dots),$$

where *action* is again dependent on the particular service. These messages typically contain an output from the protocol instance to the calling entity. There is a special output action *halt*, represented by

$$(\mathit{ID}, \texttt{out}, \texttt{halt}),$$

after which no further messages tagged with *ID* are processed by the party. When P_i generates such a message with tag *ID*, we say that P_i has *halted* instance *ID*.

We stress that in a real protocol implementation, input and output actions both do not involve any real network communication but will be mapped onto local events being generated or processed by the calling entity. But in the formal model at least some of them are generated and received by the adversary.

The third type of message generated by P_i are of the form

$$(\mathit{ID}, i, j, \dots),$$

where $1 \leq j \leq n$ denotes the index of the recipient. Such a message is called a *protocol message*; the idea is that the adversary delivers it to P_j, where it is processed by the corresponding protocol instance.

Internal Scheduling. When a party is activated by the adversary, all incoming messages are appended to a local buffer and the internal *scheduler* is invoked. It delivers messages to the protocol instance associated with the corresponding *ID*. If no protocol associated with *ID* is running yet, the scheduler buffers all arriving messages until a corresponding instance has been opened. If the protocol instance has already halted, the message is discarded.

The applicable messages in the buffer are delivered to the protocol instances as follows. For each input action *open* with a tag *ID* that has not been opened before, a new protocol instance with the specified *ID* is initialized and the scheduler remembers that it was started over the network (i.e., by the adversary).

Each opened protocol instance may execute a **wait for** operation specifying the types of messages it is accepting and the particular conditions under which it processes them. When a message is delivered to a protocol instance, the instance processes the message, potentially generating some messages, until it performs the next **wait for** operation or an explicit **halt** operation. The scheduler translates **halt** into the output action *halt* for tag *ID* and removes the instance *ID* (further messages tagged with *ID* are ignored).

The scheduler treats messages generated by an instance *ID* as follows. *Protocol messages* are simply written to the outgoing communication interface. For each input action *open*, however, a new protocol instance with the specified child *ID* is initialized, as if the message came from the network. The scheduler remembers the *ID* of the parent instance; all subsequent *input actions* from the parent addressed to the child are not written out to the network, but included directly in the buffer. Each *output action* of a sub-protocol instance *ID* is mapped directly into a corresponding internal message for its parent; *output actions* of a root protocol instance are written to the outgoing communication interface. These steps allow local activation of sub-protocols and local processing of their output to be described in terms of the single message interface.

The scheduler continues to deliver messages to protocol instances in an arbitrary order, until the buffer contains no more applicable messages. When no instance is **waiting for** any message present in the buffer, control is returned to the adversary. Some messages may remain in the buffer until the next activation because no protocol was waiting for them. Correctness and security of a protocol instance should not depend on the implementation of the scheduler, as long as it obeys these rules.

Our protocol descriptions are mostly written in reactive style, consisting simply of message handlers for which a global **wait for** operation is issued implicitly. Upon receiving an applicable message, the handler will execute some instructions, update its state, and may also perform a **wait for** operation which will block until the appropriate messages have arrived. If an instance *ID* **waits for** messages tagged with *its own ID*, it is simply a shorthand notation for the corresponding message handlers. But if an instance *ID* **waits for** output from a child instance (that has previously been opened), the scheduler delivers the output actions of the child to the parent, as mentioned before. We make the assumption that an instance **waiting for** output from an uninitialized instance triggers implicitly a corresponding *open* action, which initializes the instance.

For simplicity, we shall assume that messages are *authenticated*, which means that we restrict the adversary's behavior as follows: if P_i and P_j are honest, and the adversary delivers a protocol message M of the form $(ID, i, j, \dots)$ to P_j, then M was generated by P_i at some prior point in time. It is reasonable to build authentication into our model because it can be implemented very cheaply using standard symmetric-key cryptographic techniques [48].

3.1.1.3 Quantitative Aspects

Defining Termination. In the model with computationally bounded participants considered here, we cannot apply the notion of "eventual" termination traditionally used in distributed computing, which allows for infinite protocol runs and would make formal models of cryptographic methods with computationally bounded adversaries meaningless. Instead, we define termination of a protocol instance only to the extent that the adversary faithfully delivers messages among the honest parties (analogous to [12]). To bound the adversary's running time, we must be able to quantify the amount of work done by honest parties on behalf of a protocol. We measure the *efficiency* of a protocol for this purpose. Combined with liveness conditions (such as "validity"), restricting the amount of work implies eventual termination in the conventional sense.

Formally, our efficiency condition is based on a statistic $X(k)$ measuring the work done *by honest parties* in a multi-party protocol execution, such as "useful" computation steps or the number of generated message bits. The statistic is a non-negative value depending on the protocol and on the adversary, and is a function of the security parameter. We only allow statistics that are polynomial in the security parameter (but depending on the adversary); assume w.l.o.g. that every statistic is bounded by the running time of the adversary. In general, $X(k)$ is a discrete random variable (actually, a family $\{X(k)\}_k$ of random variables indexed by k) induced by the coin flips of the dealer, the honest parties, and the adversary. The key to defining efficiency is bounding the statistic *independently* of the adversary, i.e., in such a way that the bound depends only on the particular protocol. This will rule out trivial protocols that never terminate but always cause some work to be done without making progress.

As we consider deterministic and randomized protocols (which may not always terminate after a polynomial number of steps), we introduce two corresponding notions for polynomially bounding a statistic as follows. The probabilistic bound allows the statistic to exceed the fixed polynomial with non-negligible probability, but this probability is again independent of the adversary.

Definition 1 (Bounded Statistics). Let $T(k)$ and $U(k)$ be arbitrary fixed polynomials on the integers, independent of the adversary, and let $X(k)$ be a statistic as above (a discrete non-negative random variable induced by the adversary). We say that

1. $X(k)$ is *[deterministically] polynomially bounded* if for negligible ϵ and for all adversaries,
$$\Pr[X(k) > T(k)] \;\leq\; \epsilon(k);$$

2. $X(k)$ is *probabilistically polynomially bounded* if for negligible ϵ, for all adversaries, and for all $l \geq 1$,
$$\Pr[X(k) > lU(k) + T(k)] \;\leq\; 2^{-l} + \epsilon(k).$$

Although a probabilistically bounded statistic is not bounded by a fixed polynomial, its expected value is at most $O(U(k) + T(k))$. More precisely, the following can be shown.

Lemma 1. *Suppose $X(k)$ is a probabilistically polynomially bounded statistic of a multi-party protocol, with polynomials $T(k)$ and $U(k)$. Then the expected value of $X(k)$ is bounded by $2U(k) + T(k) + \epsilon'(k)$, for negligible ϵ'.*

Proof. Recall that the statistic is bounded by some polynomial $q(k)$ in the security parameter (depending on the adversary); thus the random variable $X(k)$ exceeds $q(k)$ with probability zero.

Set $X'(k) = (X(k) - T(k))/U(k)$; it follows $X'(k) \leq q'(k)$ for some polynomial $q'(k)$. Because $X(k)$ is probabilistically polynomially bounded, we know that $\Pr[X'(k) > l] \leq 2^{-l} + \epsilon(k)$ for all $l \geq 1$. Together with $\mathrm{E}[Y] \leq \sum_{l \geq 0} \Pr[Y > l]$ for any non-negative discrete random variable Y, it follows

$$\mathrm{E}[X'(k)] \;\leq\; \sum_{l \geq 0} \Pr[X'(k) > l] \;\leq\; \sum_{l=0}^{q'(k)} \Pr[X'(k) > l] \;\leq\; \sum_{l=0}^{q'(k)} 2^{-l} + \epsilon(k) \;\leq\; 2 + q'(k)\epsilon(k).$$

Because ϵ is negligible and by the linearity of the expectation, this implies that there is a negligible ϵ' for which $\mathrm{E}[X(k)] \leq 2U(k) + T(k) + \epsilon'(k)$. □

Communication and Message Complexity. An appropriate statistic in the above sense is the *communication complexity* of a multi-protocol; it is used in this work to define termination. Formally, the communication complexity is equal to the *bit length* of all *associated* protocol messages that *honest parties generate*. Which protocol messages are associated to a particular instance ID will vary according to the protocol type and will be noted explicitly when defining a protocol. Typically, this includes all messages with the tag ID or any tag starting with $ID|\ldots$; through the second form, also messages generated by sub-protocols on behalf of the calling protocol can be associated to an instance ID. Our protocol architecture ensures that all messages generated by honest parties are associated to some protocol.

Restricting to messages *generated* by *honest* parties seems the best one can say about a protocol in a Byzantine environment; the adversary can always deliver "junk" protocol messages to honest parties, which require some work to be read. Network bandwidth is an apparent resource that communication protocols consume, thus, measuring it seems adequate.

Alternatively, one could bound the bit length of all distinct messages delivered to one honest party that was generated by another honest party. But this is bounded by the communication complexity in the sense above.

As it is, there is no a priori restriction on the size of a payload message in our formal model. However, the communication complexity of a broadcast protocol depends on the length of such a message. For simplicity, we will therefore assume that there exists a fixed polynomial upper bound on the length of all payload messages that are contained in any input or output action message of any honest party. From this, and from the description of a particular protocol implementation, one can derive an upper bound on the maximal length of any protocol message.

Another appropriate statistic for a certain class of protocols (like Byzantine agreement, as used in [12]) is the *message complexity*, defined as the total *number* of all *associated* protocol messages that *honest parties generate*.

If the communication complexity (or also the message complexity) is polynomially bounded, the adversary *could* quickly make all honest parties terminate the protocol instance, but it is not *forced* to do so.

Modular Protocol Composition. Using the message complexity (or communication complexity) as a statistic has the advantage that it is closed under the *modular composition* of protocols as follows. According to our architecture, a higher-level protocol may invoke a sub-protocol to carry out a certain task; this appears as a one or more input actions generated by the higher-level protocol, which will start the sub-protocol as described above.

Suppose for the moment that the sub-protocol is implemented by a distributed oracle available to every party, which provides the service of the sub-protocol in an ideal and instantaneous way. We call such a protocol an *oracle protocol.* A party invokes the protocol oracle by generating a suitable message, so that this counts as one unit in the message complexity statistic.

Consider two multi-party oracle protocols A and B with respective message complexities X_A and X_B that both are both polynomially bounded. Suppose that the oracle protocol A uses an oracle for the task provided by B. Since B is implemented by the oracle, X_A counts every invocation of B by any honest party as one unit.

If we replace every oracle call by A to B by actually invoking B according to our general system model, we obtain a composed protocol AB with message complexity X_{AB}. This counts all messages that protocol A generates directly and those generated by the instances of B started on behalf of A. But because X_A and X_B are polynomially bounded, there exist fixed, polynomial upper bounds on the message complexities of A and B and also on the number of activations of protocol B (because message complexity bounds also the number of sub-protocol invocations). Thus, there exists also a polynomial upper bound on X_{AB}, independent of the adversary.

In other words, if we compose two protocols with polynomially bounded message complexities (one of them being an oracle protocol), we obtain another protocol with polynomially bounded message complexity. This extends trivially to communication complexity and, in fact, to any statistic in which invoking a sub-protocol is counted as one cost unit. So we have shown the following:

Lemma 2. *Polynomially bounded communication complexity is closed under the modular composition of protocols.*

Using a simple, but tedious calculation, one can also show that this holds for randomized protocols with probabilistically polynomially bounded communication complexity.

Lemma 3. *Probabilistically polynomially bounded communication complexity is closed under the modular composition of protocols.*

This is an important property of our notion of termination for randomized protocols and justifies the way in which we have defined it. If one would merely consider the *expected value* of a statistic for a randomized protocol, one could not draw such conclusions. For example, combining a protocol from which we only know that its expected number of rounds is constant with another one having the same property would not guarantee that the total expected number of rounds is also constant.

3.1.2 Byzantine Agreement

We give the definition of *Byzantine agreement* (or *consensus* in the crash-fault model) here as it is needed for building atomic broadcast protocols. It can be used to provide agreement on independent *transactions*.

The Byzantine agreement protocol is activated when the adversary delivers a message to P_i of the form

$$(ID, \texttt{in}, \texttt{propose}, v),$$

where $v \in \{0, 1\}$. When this occurs, we say P_i *proposes* v for transaction ID.

A party terminates the Byzantine agreement protocol (for transaction ID) by generating an output message of the form

$$(ID, \texttt{out}, \texttt{decide}, v).$$

In this case, we say P_i *decides* v for transaction ID.

Let any message with tag ID or $ID|\ldots$ that is generated by an honest party be *associated* to the agreement protocol for ID.

Definition 2 (Byzantine agreement). A protocol solves *Byzantine agreement* if it satisfies the following conditions except with negligible probability:

Validity: If all honest parties that are activated on a given ID propose v, then any honest party that terminates for ID decides v.

Agreement: If an honest party decides v for ID, then any honest party that terminates decides v for ID.

Liveness: If all honest parties have been activated on ID and all associated messages have been delivered, then all honest parties have decided for ID.

Efficiency: For every ID, the communication complexity for ID is probabilistically polynomially bounded.

This is the usual definition of validity in the literature. In Section 3.3 we introduce the weaker notion of *external validity* that is useful for certain applications. For instance, if initial values come with validating data (e.g., a digital signature) that establishes their validity in a particular context, we will require that an honest party may only decide on a value for which it has the accompanying validating data. Thus, even if all honest parties start with 0, they may still decide on 1 if they obtain the corresponding validating data for 1 during the agreement protocol.

3.1.3 Cryptographic Primitives

Apart from ordinary digital signature schemes, we use robust, non-interactive threshold signatures, threshold public-key encryption schemes, and a threshold coin-tossing protocol.

We need a *collision-free hash function* $H : \{0,1\}^* \rightarrow \{0,1\}^{k'}$ with the property that the adversary cannot generate two distinct strings x and x' such that $H(x) = H(x')$, except with negligible probability.

Another useful primitive is a cryptographically strong *pseudorandom generator* [34], denoted by $G : \{0,1\}^{k''} \rightarrow \{0,1\}^*$, that stretches a k''-bit seed by an arbitrary polynomial factor. G is a deterministic algorithm with input a random k''-bit seed such that its output is computationally indistinguishable from a uniform random string of the same length. In other words, for every efficient statistical test running in time polynomial in k, the probability that it can distinguish the output of G with a random seed from truly random bits is negligible.

Many efficient cryptographic schemes, and in particular all the threshold-cryptography protocols needed below, can be analyzed only in the so-called *random-oracle model* [4]. This refers to an idealized world where a hash function has been replaced by a truly random oracle, available to all participants. Although such proofs provide only a heuristic notion of security, the model allows to design truly practical protocols that admit a security analysis, which yields very strong evidence for their security.

3.1.3.1 Digital Signatures

A digital signature scheme [35] consists of a *key generation* algorithm, a *signing* algorithm, and a *verification* algorithm. The key generation algorithm takes as input a security parameter, and outputs a public key/private key pair. The signing algorithm takes as input that private key and a message m, and produces a signature σ. The verification algorithm takes the public key, a message m, and a putative signature σ, and outputs a bit that indicates whether it accepts or rejects the signature. A signature is considered *valid* if and only if the verification algorithm accepts. All signatures produced by the signing algorithm must be valid.

The basic security property is *unforgeability*. The attack scenario is as follows. An adversary is given the public key, and then requests the signatures on a number of messages, where the messages themselves may depend on previously obtained signatures. If at the end of the attack, the adversary can output a message m and a valid signature σ on m, such that m was not one of the messages whose signature it requested, then the adversary has successfully *forged* a signature. Security means that it is computationally infeasible for an adversary to forge a signature.

3.1.3.2 Non-Interactive Threshold Signatures

An important tool for our broadcast protocols are non-interactive threshold signatures. More precisely, we need *dual-threshold* variations as introduced by Shoup [59] and Cachin, Kursawe, and Shoup [12]. The basic idea of a dual-threshold signature scheme is that there are n parties, t of which may be corrupted. The parties hold *shares* of the secret key of a signature scheme, and may generate *shares of signatures* on individual messages. The only requirement is that κ signature shares are necessary and sufficient to construct a signature, where $t < \kappa \leq n - t$. (The standard notion of threshold schemes considers only $\kappa = t + 1$.)

More precisely, a non-interactive (n, κ, t)-dual-threshold signature scheme consists of the following parts:

- A *key generation algorithm* with input parameters k, n, κ, and t. It outputs the public key of the scheme, a private key share for each party, and a local verification key for each party.
- A *signing algorithm* with inputs a message, the public key and a private key share. It outputs a signature share on the submitted message.
- A *share verification algorithm* with inputs a message, a signature share on that message from a party P_i, along with the global public key and the local verification key of P_i. It determines if the signature share is valid.
- A *share combining algorithm* that takes as input a message and κ valid signature shares on the message, along with the public key and the verification keys, and (hopefully) outputs a valid signature on the message.
- A *signature verification algorithm* that takes as input a message and a signature (generated by the share-combining algorithm), along with the public key, and determines if the signature is valid.

The interaction takes place in the basic system model introduced above. During initialization, the dealer runs the key generation algorithm and gives each party the public key, all local verification keys, and its private key share. The adversary may submit signing requests to the honest parties for messages of its choice. Upon receiving such a request, a party computes a *signature share* for the given message using its private key share. Given κ valid signature shares from distinct parties on the same message, they may be combined into a signature on the message.

The two basic security requirements are *robustness* and *non-forgeability.* Robustness means that it is computationally infeasible for an adversary to produce κ valid signature shares such that the output of the share combining algorithm is not a valid signature. Non-forgeability means that it is computationally infeasible for the adversary to output a valid signature on a message that was submitted as a signing request to *less* than $\kappa - t$ honest parties.

A practical scheme that satisfies these definitions in the random-oracle model was proposed by Shoup [59] and is based on RSA [57]. Each signature share has essentially the size of an RSA signature and the final signature is a standard RSA signature. Our definition of a threshold signature scheme would also admit the trivial implementation of just using a set of κ ordinary signatures.

The dual-threshold scheme is used in some of our protocols, where a threshold signature with $\kappa > t + 1$ provides evidence for the fact that $\kappa - t$ honest parties have executed some steps in the protocol. A single-threshold scheme would not work here because although our system corruption model is static, the adversary may adaptively decide from which honest parties to request additional signature shares by scheduling messages accordingly.

3.1.3.3 Non-Interactive Threshold Cryptosystems

We use the definition of non-interactive threshold cryptosystems with security against adaptive chosen-ciphertext attacks put forward by Shoup and Gennaro [60]. (For ordinary public-key cryptosystems, security against adaptive chosen-ciphertext attacks is equivalent to non-malleability [23].)

A $(n, t + 1)$-threshold cryptosystem is given by the following algorithms:

- A *key generation algorithm*, taking as input k, n, and t. Outputs are the public key and a private decryption key for each party.
- An *encryption algorithm* with inputs the public key, a cleartext message $m \in \{0,1\}^*$. The algorithm outputs a ciphertext c and a label $\ell \in \{0,1\}^*$.
- A *decryption algorithm* with inputs the public key, an index $i \in \{1, \ldots, n\}$, the private key of P_i, a ciphertext c, and a label ℓ. It outputs a *decryption share* or a special symbol $\perp$ if the inputs are invalid.

- A *combination algorithm* that takes as inputs the public key, a ciphertext c, a label ℓ and a list D of decryption shares, of which some may be invalid. If D contains at least $t+1$ valid decryption shares, the algorithm outputs the cleartext m. Otherwise it returns a special symbol $\perp$.

The interaction takes place in the basic system model according to Section 3.1.1. During the initialization phase, the dealer runs the key generation algorithm and gives each party the global public key and its private key share.

Any party may run the encryption algorithm with the public key and a cleartext message to produce a ciphertext.

For decryption, a party sends the ciphertext together with the label to each party P_i, who returns a decryption share. Upon receiving enough decryption shares, the decryptor can combine them in order to obtain the cleartext.

The algorithms ensure that if a ciphertext c of a cleartext m was produced correctly by the encryption algorithm, then the recovery algorithm yields m with all but negligible probability, even if at most t decryption shares were not produced by the decryption algorithm with inputs as specified above. This property is called *robustness*.

To define security against adaptive chosen ciphertext attacks, consider the following game, played by the adversary in our basic system model with t statically corrupted parties; the keys generated by the dealer and given to the corrupted parties are seen by the adversary.

A1. The adversary interacts with the uncorrupted parties in an arbitrary fashion, feeding them ciphertext/label pairs and obtaining decryption shares.

A2. The adversary chooses two cleartexts, m_0 and m_1, and gives them to an "encryption oracle." The oracle chooses a bit b at random, encrypts m_b, and returns the resulting ciphertext c and label ℓ to the adversary.

A3. The adversary continues to interact with the uncorrupted parties, feeding them ciphertext/label pairs (c', ℓ') and receiving decryption shares, with the restriction that $(c', \ell') \neq (c, \ell)$.

A4. The adversary outputs a bit $\hat{b}$.

The threshold cryptosystem is called *secure against adaptive chosen ciphertext attack* if for any polynomial-time bounded adversary the probability that $b = \hat{b}$ exceeds 1/2 only by a negligible quantity.

A practical threshold cryptosystem according to the above definition has been presented by Shoup and Gennaro [60]. Its security is based on the computational Diffie-Hellman problem [22], and it works in the random-oracle model; a variation of it is based on the decisional Diffie-Hellman problem.

3.1.3.4 Threshold Coin-Tossing

We also need an $(n, t+1)$-threshold coin-tossing scheme. The basic idea is the same as for the other threshold primitives, but here the parties hold *shares* of a pseudorandom function F. It maps a bit string N, the name of a coin, to its value $F(N) \in \{0,1\}^{k''}$. We use a generalized coin that produces k'' random bits simultaneously; such a coin is also called a distributed pseudo-random function [50]. The parties may generate *shares of a coin value* $F(N)$ and $t+1$ shares of the same coin are both necessary and sufficient to construct the value of that coin. The generation and verification of coin shares are also non-interactive and we work in the basic system model of Section 3.1.1.

During initialization the dealer generates a global verification key, a local verification key for each party, and a secret key share for each party. The initial state information for each party consists of its secret key share and all verification keys. The secret keys implicitly define a function F mapping names to k''-bit strings.

After the initialization phase, the adversary submits *reveal requests* to the honest parties for coins of his choice. Upon receiving such a request, a party outputs a *coin share* for the given coin computed from its secret key.

The coin-tossing scheme also specifies two algorithms:

- The *share verification* algorithm takes as input the name of a coin, a share of this coin from a party P_i, along with the global verification key and the verification key of P_i, and determines if the coin share is valid.
- The *share combining* algorithm takes as input a the name N of a coin and $t+1$ valid shares of N, along with (perhaps) the verification keys, and (hopefully) outputs $F(N)$.

The security requirements are *robustness* and *pseudorandomness.* Robustness means that it is computationally infeasible for an adversary to produce a name N and κ valid shares of coin N such that the output of the share combining algorithm is not $F(N)$. To define pseudorandomness, consider the following game, played in the basic system model.

D1. The adversary interacts with the uncorrupted parties in an arbitrary fashion, obtaining shares for arbitrary coins.

D2. The adversary chooses a coin N for which it has not yet requested a coin share, and gives it to an "F-oracle." The oracle chooses a bit b at random, and returns $F(N)$ if $b = 0$ and a uniformly random k''-bit string otherwise.

D3. The adversary continues to interact with the uncorrupted parties and may obtain shares for arbitrary coins, except for N.

D4. The adversary outputs a bit $\hat{b}$.

The threshold coin-tossing scheme is *pseudorandom* if for any polynomial-time bounded adversary the probability that $b = \hat{b}$ exceeds 1/2 only by a negligible quantity.

An efficient threshold coin-tossing scheme in the random-oracle model has been presented by Cachin, Kursawe, and Shoup [12]. Although their implementation produces single-bit outputs, it can be trivially modified to generate k''-bit strings, just by using a k''-bit hash function to compute the final value. Its security is based on the computational Diffie-Hellman problem in the random-oracle model. A related scheme for a distributed pseudo-random function, with security based on the decisional Diffie-Hellman problem, has also been proposed by Naor, Pinkas, and Reingold [50].

3.2 Broadcast Primitives

In this section, we introduce two broadcast primitives, *reliable broadcast* and *consistent broadcast*, and present communication-efficient protocols for both. In terms of our definitions, reliable broadcast (the Byzantine generals problem) appears as an extension of consistent broadcast; but we introduce reliable broadcast first because it is a well-known primitive. We also introduce the notion of a *verifiable* broadcast.

3.2.1 Reliable Broadcast

Reliable broadcast provides a way for a party to send a message to all other parties. It requires that all honest parties deliver the same set of messages and that this set includes all messages broadcast by honest parties, without guaranteeing anything about the order in which messages are delivered. In the context of arbitrary faults, reliable broadcast is also known as the *Byzantine generals problem* [41].

3.2.1.1 Definition

Broadcasts are parameterized by a tag *ID*, which can also be thought of as identifying a broadcast "channel." Since many parties can potentially broadcast several payload messages with the same *ID*, we augment the tag in a reliable broadcast by the identity of the sender, j, and by a sequence number s. Then, we restrict the adversary to submit a request for reliable broadcast tagged with $ID|j|s$ to P_i only if $i = j$ and at most once for every sequence number. These requirements are easily satisfied in practice by maintaining a message counter. Instances of reliable broadcast are always identified by $ID|j|s$ so that the simple tag *ID* alone represents a "virtual channel" for reliable broadcast; its implementation uses one independent protocol instance per payload message.

A reliable broadcast protocol is activated when the adversary delivers a message to P_j of the form

$$(ID|j|s, \texttt{in}, \texttt{r-broadcast}, m),$$

with $m \in \{0,1\}^*$ and $s \in \mathbb{N}$. When this occurs, we say P_j *reliably broadcasts* m *tagged with* $ID|j|s$, or simply P_j *r-broadcasts* m. Note that only P_j is activated like this. The other parties are activated when they perform an explicit *open* action for instance $ID|j|s$ in their role as receivers; according to our convention, this occurs for instance when they **wait for** an output tagged with $ID|j|s$.

A party terminates a reliable broadcast of m tagged with $ID|j|s$ by generating an output message of the form

$$(ID|j|s, \texttt{out}, \texttt{r-deliver}, m).$$

In this case, we say P_i *reliably delivers* m *tagged with* $ID|j|s$ (or *r-delivers* for brevity).

We say that all protocol messages which are generated by honest parties have tags with prefix $ID|j|s$ are *associated* to the broadcast of m by P_j with sequence number s. Recall that this defines also the messages contributing to the communication complexity of the protocol instance $ID|j|s$.

Definition 3 (Reliable Broadcast). A protocol for *reliable broadcast* satisfies the following conditions except with negligible probability:

Validity: If an honest party has *r-broadcast* m tagged with $ID|j|s$, then all honest parties *r-deliver* m tagged with $ID|j|s$, provided all honest parties have been activated on $ID|j|s$ and the adversary delivers all associated messages.

Consistency: If some honest party *r-delivers* m tagged with $ID|j|s$ and another honest party *r-delivers* m' tagged with $ID|j|s$, then $m = m'$.

Totality: If some honest party *r-delivers* a message tagged with $ID|j|s$, then all honest parties *r-deliver* some message tagged with $ID|j|s$, provided all honest parties have been activated on $ID|j|s$ and the adversary delivers all associated messages.

Integrity: For all ID, senders j, and sequence numbers s, every honest party *r-delivers* at most one message m tagged with $ID|j|s$. Moreover, if all parties are honest, then m was previously *r-broadcast* by P_j with sequence number s.

Efficiency: For any ID, sender j, and sequence number s, the communication complexity of instance $ID|j|s$ is polynomially bounded.

Some remarks on the above definition. Recall the implicit quantification over all polynomial-time adversaries.

1. Validity ensures the liveness of a protocol, and rules out trivial protocols that do not generate any messages. One could use an equivalent, but simpler definition here, requiring that only the sender (and not all honest parties) *r-deliver* the message; but then one would have to modify this again to the present form for defining consistent broadcast below.

2. The agreement condition found in traditional definitions is split into consistency and totality. The reason for separating them is not only that they are distinct properties, but also that a reliable broadcast without a totality guarantee is a useful notion, as shown later.

3. The provision that the "adversary delivers all associated messages" is our quantitative counterpart to the traditional "eventual" delivery assumption. It can be ensured for an arbitrary adversary as follows. Suppose the adversary halts and there are yet undelivered protocol messages among honest parties (these can be inferred from a transcript of the adversary's interactions). Then using a "benign" scheduler delivering all the undelivered messages and the newly generated ones, the protocol is run until no more undelivered protocol messages exist, whereby termination in polynomial time is guaranteed by efficiency and validity.

4. Integrity may seem weak, since our model assumes authenticated links and we could hope to get the guarantee in the second clause also with t actually corrupted parties. Indeed, most reliable broadcast protocols implicitly also authenticate the sender of a message. It is possible to define the corresponding notion of an *authenticated* reliable broadcast by replacing the integrity condition above by the following.

   Authenticity: For all ID, senders j, and sequence numbers s, every honest party *r-delivers* at most one message m tagged with $ID|j|s$. Moreover, if P_j is honest, then m was previously *r-broadcast* by P_j with sequence number s.

However, we will not use authenticity in the standard definitions below because only some of our protocols provide authenticity. In particular, the protocols for reliable and for consistent broadcast provide authenticity, but not the atomic broadcast protocol.

We should note that an actual *implementation* of reliable broadcast is not needed by any of our protocols below. However, we build on the *definition* of reliable broadcast for defining other forms of broadcast. Nevertheless, we give a protocol for reliable broadcast in the next section—for completeness and to illustrate the system model and our definitions.

3.2.1.2 A Protocol for Reliable Broadcast

A message-efficient reliable broadcast protocol, denoted RBC, is given in Figure 1; it results from a small modification of Bracha's reliable broadcast protocol [10] to reduce the communication complexity.

Protocol RBC uses the hash of a payload message as a short, but unique representation for the potentially much longer message. The idea is that the payload is sent only once by the sender to all parties (similar to [54]). When a party is ready to deliver a payload message but does not yet know it, it asks an arbitrary subset of $2t+1$ parties for its contents and at least one of them will answer with the correct value.

In the description of the protocol, recall the global **wait for** condition for any message with a matching tag. Let $\perp$ denote a special value that cannot be broadcast. To implement the condition that a particular message from a party is processed only the first time it is received, one has to maintain the corresponding flags and counters, indexed by the contents of the message.

Theorem 4. *Assuming H is a collision-free hash function, Protocol* RBC *provides authenticated reliable broadcast for $n > 3t$.*

Proof. *Validity* is clear for honest senders by inspection of the protocol because all parties receive the initial `r-send` message and also $2t+1$ `r-ready` messages from honest parties, provided all associated messages are delivered. It may not hold for faulty senders, though.

For *consistency*, suppose an honest party P_i has *r-delivered* m and another honest party $P_{i'}$ has *r-delivered* $m' \neq m$ with tag $ID|j|s$. Then P_i must have received `r-ready` messages containing $d = H(m)$ from at least $t+1$ honest parties; the same holds for $P_{i'}$ with $d' = H(m')$. If $d = d'$, the adversary has created a collision in H. We assume no such collisions occur in the rest of the proof.

```
Protocol RBC for party $P_i$ and tag $ID|j|s$

Initialization:
        $\bar{m} \leftarrow \bot; \bar{d} \leftarrow \bot$
        $e_d \leftarrow 0; r_d \leftarrow 0 \qquad (d \in \{0,1\}^{k'})$

Upon receiving message $(ID|j|s, \texttt{in}, \texttt{r-broadcast}, m)$:
        send $(\texttt{r-send}, m)$ to all parties

Upon receiving message $(\texttt{r-send}, m)$ from $P_l$:
        if $j = l$ and $\bar{m} = \bot$ then
            $\bar{m} \leftarrow m$
            send $(\texttt{r-echo}, H(m))$ to all parties

Upon receiving message $(\texttt{r-echo}, d)$ from $P_l$ for the first time:
        $e_d \leftarrow e_d + 1$
        if $e_d = n - t$ and $r_d \leq t$ then
            send $(\texttt{r-ready}, d)$ to all parties

Upon receiving message $(\texttt{r-ready}, d)$ from $P_l$ for the first time:
        $r_d \leftarrow r_d + 1$
        if $r_d = t + 1$ and $e_d < n - t$ then
            send $(\texttt{r-ready}, d)$ to all parties
        else if $r_d = 2t + 1$ then
            $\bar{d} \leftarrow d$
            if $H(\bar{m}) \neq d$ then
                send $(\texttt{r-request})$ to $P_1, \ldots, P_{2t+1}$
                wait for a message $(\texttt{r-answer}, m)$ such that $H(m) = \bar{d}$
                $\bar{m} \leftarrow m$
            output $(ID|j|s, \texttt{out}, \texttt{r-deliver}, \bar{m})$

Upon receiving message $(\texttt{r-request})$ from $P_l$ for the first time:
        if $\bar{m} \neq \bot$ then
            send $(\texttt{r-answer}, \bar{m})$ to $P_l$
```

Figure 1: Protocol RBC for authenticated reliable broadcast (or the Byzantine generals problem) adopted from Bracha [10].

An honest party generates an `r-ready` message for d only if it has received $n - t$ `r-echo` messages containing d or $t + 1$ `r-ready` messages already containing d. Thus, at least one honest party has sent an `r-ready` message containing d upon receiving $n - t$ `r-echo` messages; at most t of them are from corrupted parties. Similarly, some honest party must have received $n - t$ `r-echo` messages containing d'. Thus, there are at least $2(n - t) \geq n + t + 1$ `r-echo` messages with tag $ID|j|s$ and at least $n - t + 1$ among them from honest parties. But no honest party generates more than one such message by the protocol.

To establish *totality*, note that if some honest P_i delivers $\bar{m}$, then it has received the message $(\texttt{r-ready}, \bar{d})$ from $2t + 1$ different parties. Therefore, at least $t + 1$ honest parties have sent `r-ready` with $ID|j|s$ and $\bar{d} = H(\bar{m})$, which will be received by all honest parties (assuming the adversary delivers all messages). Thus, all honest parties will send the corresponding `r-ready` message and any other party P_l will receive $2t + 1$ of them. If P_l already knows m' with $H(m') = \bar{d}$, it outputs that.

Otherwise, P_l will send an `r-request` to $2t + 1$ parties and wait for an `r-answer` satisfying $H(m') = \bar{d}$. Observe that there is at least one honest party who has sent an `r-ready` message containing $\bar{d}$ upon receiving $n - t$ corresponding `r-echo` messages. Thus, there are at least $n - 2t$ honest parties who sent `r-echo` and know some m' such that $H(m') = \bar{d}$. Sending the `r-request` to $2t + 1$ parties ensures that at least one of them receives and answers it, provided all messages are delivered.

For *integrity*, the uniqueness of the *r-delivered* message is clear from the protocol. If the sender P_j of message with sequence number s is honest, then at most t parties will send `r-echo` messages for tag $ID|j|s$ with $m' \neq m$. Thus, no uncorrupted party generates an `r-ready` message with d different from $H(m)$ and no uncorrupted party outputs m'. Actually, the protocol also satisfies *authenticity* because honest parties process `r-send` messages only from the sender indicated by the `r-echo` message.

It is easy to see that the protocol satisfies *efficiency* for any sender. □

Note that collecting $n-t$ `r-echo` messages is needed for totality (because `r-request` messages are sent to only $2t + 1$ parties), but for consistency alone, this could be relaxed to $\lceil \frac{n+t+1}{2} \rceil$ `r-echo` messages.

The message complexity of Protocol RBC is $O(n^2)$. If messages are delivered faithfully by a "benign" scheduler and no faults occur, then its communication complexity is only $O(n^2k' + n|m|)$ for broadcasting a single message m, where k' is the length of a hash value. However, the adversary can delay the `r-send` messages for some parties and increase the communication complexity. Since there are at most t honest parties who issue an `r-request` by the argument above to establish totality, m is transmitted $O(t^2)$ times and the overall communication complexity is $O(n^2k' + n|m| + t^2|m|)$, or $O(n^2|m|)$ with maximal resilience.

Contrast this with the standard form of Bracha's broadcast that requires bit complexity $\Omega(n^2|m|)$, even in executions without faults. Under optimal circumstances, Protocol RBC needs to transmit m only once per party in the system.

3.2.2 Verifiable Broadcast

A party P_i that has delivered a payload message using reliable broadcast may want to inform another party P_j about this. Such information might be useful to P_j if it has not yet delivered the message, but can exploit this knowledge somehow, in particular since P_j is guaranteed to deliver the same message later by the agreement property. In a standard reliable broadcast, such as the protocol from the previous section, however, this knowledge cannot be transferred in a verifiable way.

We formalize this property of a broadcast protocol here because it is useful in our application below, and call it *verifiability.* Informally, it means this: when P_j claims that it is not yet in a state to deliver a particular payload message m, then P_i can reply with a single protocol message and when P_j processes this, it will deliver m immediately and terminate the corresponding broadcast.

Definition 4 (Verifiability). A broadcast protocol is called *verifiable* if the following holds, except with negligible probability: When an honest party has delivered m tagged with ID, then it can produce a single protocol message M that it may send to other parties such that any other honest party will deliver m tagged with ID upon receiving M (provided the other party has not already delivered m).

We call M the message that *completes* the verifiable broadcast. This notion implies that there is a predicate V_{ID} that the receiving party can apply to an arbitrary bit string for checking if it constitutes a message that completes a verifiable broadcast tagged with ID.

Protocol RBC could be made verifiable by adding a digital signature to the *r-ready* messages (this idea goes back to Pease, Shostak, and Lamport [51]). But verifiability is more useful in connection with weaker protocols than reliable broadcast; for example, in the consistent broadcast introduced next.

3.2.3 Consistent Broadcast

The totality property of reliable broadcast is rather expensive to satisfy; it is the main reason why most protocols for reliable broadcast need on the order of n^2 messages. For some applications, however, totality is not necessary and can be ensured by other means, as long as consistency and integrity are satisfied. We call the resulting notion *consistent broadcast* and discuss it in this section.

Several protocols for consistent broadcast have been proposed by Reiter et al. [54, 45]. To ensure agreement (i.e., totality) for delivered messages, these protocols are complemented by an external stability mechanism from which parties learn about the existence of messages they have not yet delivered. No such general mechanism is assumed here, but the parties may learn that also from an application.

3.2.3.1 Definition

The same restrictions on the adversary apply as for reliable broadcast. A consistent broadcast protocol is activated when the adversary delivers a message to P_j of the form

$$(ID|j|s, \texttt{in}, \texttt{c-broadcast}, m),$$

with $m \in \{0,1\}^*$ and $s \in \mathbb{N}$. When this occurs, we say P_j *consistently broadcasts* m *tagged with* $ID|j|s$.

A party terminates a consistent broadcast of m tagged with $ID|j|s$ by generating an output message of the form

$$(ID|j|s, \texttt{out}, \texttt{c-deliver}, m).$$

In this case, we say P_i *consistently delivers* m *tagged with* $ID|j|s$. To distinguish consistent broadcast from other forms of broadcast, we will sometimes use the terms *c-broadcast* and *c-deliver.*

All protocol messages generated by honest parties and tagged with $ID|j|s$ are associated to the broadcast of m by P_j with sequence number s.

Definition 5 (Consistent Broadcast). A protocol for *consistent broadcast* is a protocol for reliable broadcast that does not necessarily satisfy *totality.*

In other words, consistent broadcast makes no provisions that two parties do deliver the payload message, but maintains consistency among the actually delivered messages with the same senders and sequence numbers.

The notion of an authenticated consistent broadcast can be defined similarly to authenticated reliable broadcast, replacing the integrity condition by authenticity.

3.2.3.2 A Protocol for Verifiable Consistent Broadcast

Protocol VCBC implements verifiable consistent broadcast and is described in Figure 2. It uses a non-interactive $(n, \lceil \frac{n+t+1}{2} \rceil, t)$-dual-threshold signature scheme $\mathcal{S}_1$ with verifiable shares according to Section 3.1.3.2. Recall that all messages are authenticated according to our basic system model.

The protocol is based on the "echo broadcast" of Reiter [54], but uses a threshold signature to decrease the communication complexity. The idea behind it is that the sender broadcasts the message to all parties and hopes for $\lceil \frac{n+t+1}{2} \rceil$ parties to sign it as "witnesses" to guarantee consistency. The signature shares are then collected by the sender and combined to a threshold signature on the message; it then sends the signature all parties. After receiving the message together with a valid signature, a party delivers it immediately.

Because a party may forward the message and the signature to other parties, the protocol is also verifiable according to Definition 4. The corresponding interface is implemented by the `c-request` and `c-answer` messages, which are not otherwise used by the protocol.

The consistency property of the protocol is based on the following lemma.

Lemma 5. *For all senders j, sequence numbers s, and strings ID, it is infeasible for the adversary in Protocol* VCBC *to create valid $\mathcal{S}_1$-signatures on the strings $(ID|j|s, \texttt{c-ready}, m)$ and $(ID|j|s, \texttt{c-ready}, m')$ with $m \neq m'$.*

Proof. Suppose not. Then, assuming $\mathcal{S}_1$ is secure, there are at least $\lceil \frac{n+t+1}{2} \rceil - t$ signature shares from distinct honest parties on a message containing $ID|j|s$ and m and at least as many from honest parties on the message containing $ID|j|s$ and m'. In total, there are $n + t + 1 - 2t = n - t + 1$ or more shares generated by honest parties containing $ID|j|s$. Since there are only $n - t$ honest parties, at least one honest party has signed two different messages with the same sender j and sequence number s, which is impossible according to the protocol. □

Theorem 6. *Assuming $\mathcal{S}_1$ is a secure $(n, \lceil \frac{n+t+1}{2} \rceil, t)$-dual-threshold signature scheme, Protocol* VCBC *provides verifiable and authenticated consistent broadcast for $n > 3t$.*

Proof. Validity for an honest sender is obvious from the construction of the protocol since all honest parties generate a signature share on m as soon as they receive an `c-send`

```
Protocol VCBC for party $P_i$ and tag $ID|j|s$

INITIALIZATION:
        $\bar{m} \leftarrow \perp; \bar{\mu} \leftarrow \perp$
        $W_d \leftarrow \emptyset; r_d \leftarrow 0 \qquad (d \in \{0,1\}^{k'})$

UPON RECEIVING MESSAGE $(ID|j|s, \texttt{in}, \texttt{c-broadcast}, m)$:
        send $(\texttt{c-send}, m)$ to all parties

UPON RECEIVING MESSAGE $(\texttt{c-send}, m)$ FROM $P_l$:
        if $j = l$ and $\bar{m} = \perp$ then
            $\bar{m} \leftarrow m$
            compute an $\mathcal{S}_1$-signature share $\nu$ on $(ID|j|s, \texttt{c-ready}, H(m))$
            send $(\texttt{c-ready}, H(m), \nu)$ to $P_j$

UPON RECEIVING MESSAGE $(\texttt{c-ready}, d, \nu_l)$ FROM $P_l$ FOR THE FIRST TIME:
        if $i = j$ and $\nu_l$ is a valid $\mathcal{S}_1$-signature share then
            $W_d \leftarrow W_d \cup \{\nu_l\}$
            $r_d \leftarrow r_d + 1$
            if $r_d = \lceil \frac{n+t+1}{2} \rceil$ then
                combine the shares in $W_d$ to an $\mathcal{S}_1$-threshold signature $\mu$
                send $(\texttt{c-final}, d, \mu)$ to all parties

UPON RECEIVING MESSAGE $(\texttt{c-final}, d, \mu)$:
        if $H(\bar{m}) = d$ and $\bar{\mu} = \perp$ and $\mu$ is a valid $\mathcal{S}_1$-signature then
            $\bar{\mu} \leftarrow \mu$
            output $(ID|j|s, \texttt{out}, \texttt{c-deliver}, \bar{m})$


Implementation of verifiability property

UPON RECEIVING MESSAGE $(\texttt{c-request})$ FROM $P_l$:
        if $\bar{\mu} \neq \perp$ then
            send $(\texttt{c-answer}, \bar{m}, \bar{\mu})$ to $P_l$

UPON RECEIVING MESSAGE $(\texttt{c-answer}, m, \mu)$ FROM $P_l$:
        if $\bar{\mu} = \perp$ and $\mu$ is a valid $\mathcal{S}_1$-signature on $(ID|j|s, \texttt{c-ready}, H(m))$ then
            $\bar{\mu} \leftarrow \mu$
            $\bar{m} \leftarrow m$
            output $(ID|j|s, \texttt{out}, \texttt{c-deliver}, \bar{m})$
```

Figure 2: Protocol VCBC for verifiable and authenticated consistent broadcast.

message containing m. Since at least $\lceil \frac{n+t+1}{2} \rceil$ honest parties return them to the sender, it can combine them to a valid signature and *c-deliver* the message.

The *consistency* property follows directly from Lemma 5 because an honest party *c-delivers* a payload message only after verifying the corresponding threshold signature.

Integrity follows directly from Lemma 5 together with the logic of the protocol, where $\bar{\mu} \neq \perp$ is used to represent the state in which $\bar{m}$ has already been *c-delivered.* The protocol provides also *authenticity* because honest parties process `c-send` messages only from the sender indicated by the message.

Finally, *efficiency* is straightforward to verify and *verifiability* is ensured by the `c-answer` protocol message, which is generated upon receiving a suitable `c-request`. □

The message complexity of Protocol VCBC is $O(n)$ and its bit complexity is $O(n(|m|+K))$, assuming the length of a threshold signature and a signature share is at most K bits.

3.3 Validated Byzantine Agreement

The standard notion of Byzantine agreement implements a binary decision and can guarantee a particular outcome only if *all* honest parties propose the same value. We introduce in this section a weaker validity condition, called *external validity*, which relaxes the standard validity condition and generalizes to decisions on a value from an arbitrarily large set. It requires that the decided value satisfies a global predicate that is determined by the particular application and known to all parties. Each party adds some validation data to the proposed value, which serves as the proof for its validity. Typically, this consists of a digital signature that can be verified by all parties. The agreement protocol then returns to the caller not only the decision value, but also the corresponding validation data—the caller might need this information if it did not know it before. The standard validity condition is the special case of a trivially true predicate.

Validated Byzantine agreement generalizes the primitive of *agreement on a core set* [6, 7], which is used in the information-theoretic model for a similar purpose. Validated Byzantine agreement also generalizes the notion of *interactive consistency* [26] to the Byzantine model, which requires agreement on a vector of n values, one from each party.

Another related problem is *set agreement* [17], in which the agreement condition is relaxed so that the output of each party is contained in a small, global set. Although there exists a considerable literature on this problem, it cannot be used for our applications because it gives only an approximation of agreement.

3.3.1 Definition

Suppose there is a global polynomial-time computable predicate Q_{ID} known to all parties, which is determined by an external application. Each party may propose a value v together with a proof π that should satisfy Q_{ID}. The agreement domain is not restricted to binary values.

A validated Byzantine agreement protocol is activated by a message of the form

$$(ID, \texttt{in}, \texttt{v-propose}, v, \pi),$$

where $v \in \{0,1\}^*$ and $\pi \in \{0,1\}^*$. When this occurs, we say P_i *proposes* v *validated by* π for transaction ID. We assume the adversary activates all honest parties on a given ID at most once and, w.l.o.g., honest parties propose values with proofs that satisfy Q_{ID}.

A party terminates a validated Byzantine agreement protocol by generating a message of the form

$$(ID, \texttt{out}, \texttt{v-decide}, v, \pi).$$

In this case, we say P_i *decides* v *validated by* π for transaction ID.

We say that any protocol message with tag ID that was generated by an honest party is *associated* to the validated Byzantine agreement protocol for ID. An agreement protocol may also invoke sub-protocols for low-level broadcasts or for Byzantine agreement; in this case, all messages associated to those protocols that are started *on behalf* of the validated agreement protocol are associated to ID as well (such messages have tags with prefix $ID|\ldots$).

Definition 6 (Validated Byzantine Agreement). A protocol solves *validated Byzantine agreement* with predicate Q_{ID} if it satisfies the following conditions except with negligible probability:

External Validity: Any honest party that terminates for ID decides v validated by π such that $Q_{ID}(v, \pi)$ holds.

Agreement: If some honest party decides v for ID, then any honest party that terminates decides v for ID.

Liveness: If all honest parties have been activated on ID and all associated messages have been delivered, then all honest parties have decided for ID.

Efficiency: For every ID, the communication complexity for ID is probabilistically polynomially bounded.

In other words, honest parties may propose all different values and the decision value may have been proposed by a corrupted party, as long as honest parties obtain the corresponding validation during the protocol. Note that agreement, liveness, and efficiency are the same as in the definition of ordinary, binary Byzantine agreement.

Another variation of the validity condition is that an application may prefer one decision value over others. Such an agreement protocol may be *biased* and *always* output the preferred value in cases where other values would have been valid as well.

For binary validated agreement, we will need a protocol that is biased towards 1 below. Its purpose is to detect whether there is a validation for 1, so it suffices to guarantee termination with output 1 if $t+1$ honest parties know the corresponding information at the outset. A *binary validated Byzantine agreement protocol biased towards 1* is a protocol for validated Byzantine agreement on values in $\{0,1\}$ such that the following condition holds:

Biased External Validity: If at least $t+1$ honest parties propose 1, then any honest party that terminates for *ID* decides 1.

We describe two related protocols for multi-valued validated Byzantine agreement below: Protocol VBA, described in Section 3.3.3, needs $O(n)$ rounds and invokes $O(n)$ binary agreement sub-protocols; this can be improved to a constant expected number of rounds, resulting in Protocol VBAconst, which is described in Section 3.3.4. But first we discuss the binary case.

3.3.2 Protocols for Binary Agreement

Binary asynchronous Byzantine agreement protocols can easily be adapted to external validity. For example, in the protocol of Cachin, Kursawe, and Shoup [12] one has to “justify” the pre-votes of round 1 with a valid π. The logic of the protocol guarantees that either a decision is reached immediately or the validations for 0 and for 1 are seen by all parties in the first two rounds.

Furthermore, the protocol can be biased towards 1 by modifying the coin such that it always outputs 1 in the first round.

3.3.3 A Protocol for Multi-valued Agreement

We describe Protocol VBA that implements multi-valued validated Byzantine agreement.

The basic idea of the validated agreement protocol is that every party proposes its value as a candidate value for the final result. One party whose proposal satisfies the validation predicate is then selected in a sequence of binary Byzantine agreement protocols and this value becomes the final decision value. More precisely, the protocol consists of the following steps (see Figure 3).

Echoing the proposal (lines 1–4): Each party P_i *c-broadcasts* the value that it proposes to all other parties using verifiable authenticated consistent broadcast. This ensures that all honest parties obtain the same proposal value for any particular party, even if the sender is corrupted. Then P_i waits until it has received $n - t$ proposals satisfying Q_{ID} before entering the agreement loop.

Agreement loop (lines 5–20): One party is chosen after another, according to a fixed permutation Π of $\{1, \ldots, n\}$. Let a denote the index of the party selected in the current round (P_a is called the "candidate"). Each party P_i carries out the following steps for P_a:

1. Send a `v-vote` message to all parties containing 1 if P_i has received P_a's proposal (including the proposal in the vote) and 0 otherwise (lines 6–11).
2. Wait for $n - t$ `v-vote` messages, but do not count votes indicating 1 unless a valid proposal from P_a has been received—either directly or included in the `v-vote` message (lines 12–13).
3. Run a *binary* validated Byzantine agreement biased towards 1 to determine whether P_a has properly broadcast a valid proposal. Vote 1 if P_i has received a valid proposal from P_a and validate this by the protocol message that completes the verifiable broadcast of P_a's proposal. Otherwise, if P_i has received $n - t$ `v-vote` messages containing 0, vote 0; no validation data is needed here. If the agreement decides 1, exit from the loop (lines 14–20).

Delivering the chosen proposal (lines 21–24): If P_i has not yet *c-delivered* the broadcast by the selected candidate, obtain the proposal from the validation returned by the Byzantine agreement.

The full protocol is shown in Figure 3.

An obvious optimization of Protocol VBA is based on the observation that in most cases, adding P_a's proposal in ρ to a `v-vote` message is not necessary. If this is omitted,

Protocol VBA for party P_i, tag ID, and validation predicate Q_{ID}

LET $V_{ID|a}(v, \rho)$ BE THE FOLLOWING PREDICATE:

$$V_{ID|a}(v,\rho) \;\equiv\; (v = 0) \textbf{ or } \big(v = 1 \textbf{ and } \rho \text{ completes the verifiable authenticated c-broadcast of a message } (\texttt{v-echo}, w_a, \pi_a) \text{ with tag } ID|a|0 \text{ such that } Q_{ID}(w_a,\pi_a) \text{ holds}\big)$$

UPON RECEIVING MESSAGE $(ID, \texttt{in}, \texttt{v-propose}, w, \pi)$:

```
1: verifiably authenticatedly c-broadcast message $(\texttt{v-echo}, w, \pi)$ tagged with $ID|\texttt{vcbc}|i|0$
2: $w_j \leftarrow \perp; \pi_j \leftarrow \perp \qquad (1 \le j \le n)$
3: wait for $n-t$ messages $(\texttt{v-echo}, w_j, \pi_j)$ to be c-delivered with tag $ID|\texttt{vcbc}|j|0$
      from distinct $P_j$ such that $Q_{ID}(w_j, \pi_j)$ holds
4: $l \leftarrow 0$
5: repeat
6:   $l \leftarrow l + 1; a \leftarrow \Pi(l)$
7:   if $w_a = \perp$ then
8:     send the message $(\texttt{v-vote}, a, 0, \perp)$ to all parties
9:   else
10:    let $\rho$ be the message that completes the c-broadcast with tag $ID|\texttt{vcbc}|a|0$
11:    send the message $(\texttt{v-vote}, a, 1, \rho)$ to all parties
12:  $u_j \leftarrow \perp; r_j \leftarrow \perp \qquad (1 \le j \le n)$
13:  wait for $n-t$ messages $(\texttt{v-vote}, a, u_j, \rho_j)$ from distinct $P_j$ such
       that $V_{ID|a}(u_j, \rho_j)$ holds
14:  if there is some $u_j = 1$ then
15:    $v \leftarrow 1$; $\rho \leftarrow \rho_j$
16:  else
17:    $v \leftarrow 0$; $\rho \leftarrow \perp$
18:  propose $v$ validated by $\rho$ for $ID|a$ in binary validated Byzantine agreement
       biased towards 1, with predicate $V_{ID|a}$
19:  wait for the agreement protocol to decide some $b$ validated by $\sigma$ for $ID|a$
20: until $b = 1$
21: if $w_a = \perp$ then
22:   use $\sigma$ to complete the verifiable authenticated c-broadcast with tag $ID|\texttt{vcbc}|a|0$
        and c-deliver $(\texttt{v-echo}, w_a, \pi_a)$
23: output $(ID, \texttt{out}, \texttt{v-decide}, w_a, \pi_a)$
24: halt
```

Figure 3: Protocol VBA for multi-valued validated Byzantine agreement.

then the code for P_i to receive `v-vote` messages has to be modified as follows. If a `v-vote` from P_j indicates 1 but P_i has not yet received P_a's proposal, ignore the vote and ask P_j to supply P_a's proposal (by sending it the message $(ID|\texttt{vcbc}|a|0, \texttt{c-request})$). The `v-vote` by P_j is only taken into account after $(\texttt{v-echo}, w_a, \pi_a)$ has been *c-delivered* with tag $ID|\texttt{vcbc}|a|0$ such that $Q_{ID}(w_a, \pi_a)$ holds; however, it may still be that enough votes indicating 0 from other parties are received before that.

Lemma 7. *In Protocol* VBA*, the adversary can cause at most* $2t$ *iterations of the agreement loop.*

Proof. The proof works by counting the total number A of `v-vote` messages containing 0 that are generated by honest parties (over all iterations of the agreement loop).

Since every honest party has received a valid proposal from $n - t$ parties in the `v-echo` broadcasts, it will generate `v-vote` messages containing 0 for at most t proposing parties. Thus, $A \leq t(n - t)$.

Note that for the binary Byzantine agreement protocol to decide 0 for a particular a and to cause one more iteration of the loop, at least $n-2t$ honest parties must propose 0 for the binary agreement (otherwise, there would be $t + 1$ or more honest parties proposing 1 and the binary agreement protocol would terminate with 1, as it is biased towards 1). Since honest parties only propose 0 if they have received $n-t$ `v-vote` messages containing 0, there must be at least $n - 2t$ honest parties who have generated a `v-vote` message containing 0 in this iteration.

Let R denote the number of iterations of the loop where the binary agreement protocol decides 0. From the preceding argument, we have $A \geq R(n - 2t)$.

Combining these two bounds on A, we obtain $R(n - 2t) \leq (n - t)t$, or equivalently,

$$R \ \leq \ t + \frac{t^2}{n - 2t}.$$

Using $n - 2t \geq t + 1$, this can be simplified to $R \leq t + \frac{t^2}{t+1}$ and further to $R < 2t$. Thus, the binary agreement decides 1 at the latest in iteration $R + 1$ of the loop and the lemma follows. □

Theorem 8. *Given a protocol for biased binary validated Byzantine agreement and a protocol for verifiable authenticated consistent broadcast, Protocol* VBA *provides multi-valued validated Byzantine agreement for* $n > 3t$.

Proof. We have to establish *external validity*, *agreement*, *liveness*, and *efficiency*.

External validity follows because every honest party that proposes 1 in the agreement on party P_a has verified that Q_{ID} holds for w_a and π_a. Thus, by the standard validity condition for the binary Byzantine agreement, the decision is 0 if Q_{ID} does not hold.

For *agreement*, note that the properties of the *binary* validated Byzantine agreement protocol ensure that all parties terminate the loop with the same a. By the consistency property of consistent broadcast, all honest parties obtain the same values w_a and π_a from the broadcast tagged with $ID|\mathtt{vcbc}|a|0$. Thus, they output the same w_a.

Liveness holds by inspection of the protocol.

Efficiency follows from Lemma 3 together with Lemma 7 because there are at most $2t$ binary agreement sub-protocols invoked for a particular ID. □

The message complexity of Protocol VBA is $O(tn^2)$ if Protocol VCBC is used for verifiable consistent broadcast and the binary validated Byzantine agreement is implemented according to Section 3.3.2.

If all parties propose v and π that are together no longer than L bits, the communication complexity in the above case is $O(n^2(tK + L))$, assuming the length of a threshold signature and a signature share is at most K bits. For a constant fraction of corrupted parties, however, both values are cubic in n. As shown next, the expected message complexity can be reduced to a quadratic expression in n.

3.3.4 A Constant-round Protocol for Multi-valued Agreement

In this section we present Protocol VBAconst, which is an improvement of the protocol in the previous section that guarantees termination within a constant expected number of rounds. The drawback of Protocol VBA above is that the adversary knows the order Π in which the parties search for an acceptable candidate, i.e., one that has broadcast a valid proposal. Although at least one third of all parties are guaranteed to be accepted, as shown above, the adversary can choose the corruptions and schedule messages such that none of them is examined early in the agreement loop.

The remedy for this problem is to choose Π randomly during the protocol *after* making sure that enough parties are already committed to their votes on the candidates. This is achieved in two steps. First, one round of commitment exchanges is added before the agreement loop. Each party must commit to the votes that it will cast by broadcasting the identities of the $n - t$ parties from which it has received valid `v-echo` messages (using at least authenticated consistent broadcast). Honest parties will later only accept `v-vote` messages that are consistent with the commitments made before. The second step is to determine the permutation Π using a threshold coin-tossing scheme that outputs a random, unpredictable value after enough votes are committed. Taken together, these steps ensure that the fraction of parties which are guaranteed to be accepted are distributed randomly in Π, causing termination in a constant expected number of rounds.

The details of Protocol VBAconst are described in Figure 4 as modifications to Protocol VBA.

Protocol VBAconst for party P_i, tag ID, and validation predicate Q_{ID}

Modify Protocol VBA for party P_i, tag ID, and validation predicate Q_{ID} as follows:

1. Initialize and distribute the shares for an $(n, t+1)$-threshold coin-tossing scheme $\mathcal{C}_1$ with k''-bit outputs during system setup. Recall that this defines a pseudorandom function F. Let G be a pseudorandom generator according to Section 3.1.3.

2. Include the following instructions between lines 3 and 4 of Protocol VBA, before entering the agreement loop:

```
1: $c_j \leftarrow \begin{cases} 1 & \text{if } w_j \neq \perp \\ 0 & \text{otherwise} \end{cases} \qquad (1 \leq j \leq n)$
2: $C \leftarrow [c_1, \ldots, c_n]$
3: *authenticatedly c-broadcast* the message $(\texttt{v-commit}, C)$ tagged with $ID|\texttt{cbc}|i|0$
4: $C_j \leftarrow \perp \qquad (1 \leq j \leq n)$
5: **wait for** $n-t$ messages $(\texttt{v-commit}, C_j)$ to be *c-delivered* with tag $ID|\texttt{cbc}|j|0$
     such that at least $n-t$ entries in $C_j$ are 1
6: generate a *coin share* $\gamma$ of the coin $ID|\texttt{vba}$ and send the message $(\texttt{v-coin}, \gamma)$
     to all parties
7: **wait for** $t+1$ $\texttt{v-coin}$ messages containing shares of the coin $ID|\texttt{vba}$ and
     combine these to get the value $S = F(ID|\texttt{vba}) \in \{0,1\}^{k''}$
8: choose a random permutation $\Pi$, using the pseudorandom generator $G$ with seed
   $S$.
```

3. Modify the condition for accepting `v-vote` messages (line 13) inside the agreement loop such that $(\texttt{v-vote}, a, 0, \perp)$ from P_j is accepted only if C_j is known and $C_j[a] = 0$. (This involves also waiting for additional messages $(\texttt{v-commit}, C_j)$ to be *c-delivered* as above.)

Figure 4: Protocol VBAconst for multi-valued validated Byzantine agreement.

To analyze the protocol, we consider the state of the system at the point in time when the first honest party P_i reveals its coin share. The crucial observation is that $n-t$ "early committing" parties are committed to their 0-votes at this point because P_i has delivered the corresponding broadcasts. We are now going to investigate the number of candidates that can be rejected by the adversary, by making the binary Byzantine agreement decide 0, and the number of iterations of the agreement loop.

Lemma 9. *Let $\mathcal{A} \subseteq \{1, \ldots, n\}$ denote the set of parties that garner* less *than $n - 2t$ commitments to 0-votes from the early committers, and suppose Π is an ideal, random permutation of $\{1, \ldots, n\}$. Then, except with negligible probability,*

1. *for every $a \in \mathcal{A}$, the binary agreement protocol on $ID|a$ will decide 1;*
2. *$|\mathcal{A}| > n - 2t$;*
3. *there exists a constant $\beta > 1$ such that for all $f \geq 1$,*

$$\Pr\Big[\big(\Pi(1) \notin \mathcal{A}\big) \wedge \cdots \wedge \big(\Pi(f) \notin \mathcal{A}\big)\Big] \;\leq\; \beta^{-f}.$$

Proof. In order for the binary agreement on $ID|a$ to decide 0, there must be some honest party who proposes 0. By the instructions for computing v, it must have received $n - t$ `v-vote` messages containing 0 that are consistent with the commitments made by their issuers. But since there are only n distinct parties, at least $n - 2t$ of those 0-votes must come from early committers, which is not the case for any $a \in \mathcal{A}$. This proves the first claim.

To establish the second claim, let A denote the total number of commitments to 0-votes cast by early committers. Since every early committer may commit to voting 0 for at most t parties, we have $A \leq t(n-t)$. On the other hand, observe that $A \geq (n-|\mathcal{A}|)(n-2t)$ by the definition of $\mathcal{A}$.

Observe that these bounds on A are the same as in Lemma 7 with $R = n - |\mathcal{A}|$. Using the same argument, it follows $|\mathcal{A}| > n - 2t$.

The third claim follows now because $|\mathcal{A}|$ is at least a constant fraction of n and thus, there is a constant $\beta > 1$ such that $\Pr[\Pi(i) \notin \mathcal{A}] \leq 1/\beta$ for all $1 \leq i \leq f$. Since the probability of the f first elements of Π jointly satisfying the condition is no larger than for f independently and uniformly chosen values, we obtain

$$\Pr\Big[\big(\Pi(1) \notin \mathcal{A}\big) \wedge \cdots \wedge \big(\Pi(f) \notin \mathcal{A}\big)\Big] \;\leq\; \beta^{-f}.$$

□

Lemma 10. *Assuming $\mathcal{C}_1$ is a secure threshold coin-tossing scheme and G is a pseudo-random generator, there is a constant $\beta > 1$ such that for all $f \geq 1$, the probability of any honest party performing f or more iterations of the agreement loop is at most $\beta^{-f} + \epsilon$, where ϵ is negligible.*

Proof. This can be shown by a standard hybrid argument, where one makes a series of small modifications to transform an idealized system into the real system, argues that each change affects the adversary only with negligible probability, and then concludes that the real system behaves just like the idealized system with all but negligible probability.

The "hybrid systems" are defined by running the system

(1) with a truly random permutation Π,

(2) with the output of G replaced by truly random bits, and Π computed from that,

(3) with $F(ID|\texttt{vba})$ replaced by a random bit string, but G being a pseudorandom generator according to the protocol, and Π computed from the output of G,

(4) with F, G, and Π computed according to the protocol.

In all cases, we define a statistical test by letting the adversary run the system until the first honest party is about to release its share of the coin $ID|\texttt{vba}$, and then F, G, and Π are determined. Note that the set of early committers is defined and the set $\mathcal{A}$ (of Lemma 9) can be computed at this point. The statistical test simply outputs 0 if $\Pi(i) \notin \mathcal{A}$ for all $1 \leq i \leq f$ and 1 otherwise.

We now analyze the behavior of the statistical test.

Case (1) above corresponds to the idealized system in Lemma 9, which implies that the test outputs 0 at most with probability β^{-f}.

In case (2) above, the permutation is generated from truly random bits with uniform distribution. This can be done using an algorithm that always terminates in a polynomial number of steps such that the output permutation is statistically close to a random permutation. The behavior of any polynomial-time adversary will not be changed by this, except with negligible probability.

Cases (2) and (3) above can be mapped to the definition of a pseudorandom generator. But if G is secure, the statistical test will not be able to distinguish between them with more than negligible probability.

Finally, the difference between (3) and (4) corresponds to game C1–C4 in the definition of the coin F. Assuming F is pseudorandom, this cannot induce more than a negligible difference in the behavior of the statistical test.

In conclusion, we obtain that no polynomial-time statistical test can distinguish between (1) and (4) and therefore the conclusions of Lemma 9 apply also to the real protocol except with negligible probability. Since honest parties go through more than

f iterations of the agreement loop only if the first f elements of Π are not in $\mathcal{A}$, this probability is at most β^{-f} plus some negligible quantity. □

Theorem 11. *Given a protocol for biased binary validated Byzantine agreement and a protocol for verifiable consistent broadcast, Protocol* VBAconst *provides multi-valued validated Byzantine agreement for $n > 3t$ and invokes a constant expected number of binary Byzantine agreement sub-protocols.*

Proof. Since we have not changed the way in which binary agreement sub-protocols are invoked from Protocol VBA, we only have to show *liveness* and *efficiency* for the modified protocol.

Liveness holds because all $n - t$ honest parties broadcast correctly constructed commitments and therefore, enough valid `v-commit` and `v-vote` messages are guaranteed to be received in line 13 of the original protocol.

Efficiency follows from Lemma 3 together with Lemma 10 above, because honest parties generate a polynomial number of messages in each iteration of the agreement loop. □

The expected message complexity of Protocol VBAconst is $O(n^2)$ if Protocol VCBC is used for consistent verifiable broadcast and the binary validated Byzantine agreement is implemented according to Section 3.3.2.

If all parties propose v and π that are together no longer than L bits, the expected communication complexity in the above case is $O(n^3 + n^2(K + L))$, assuming a digital signature is K bits. The n^3-term, which results from broadcasting the commitments, has actually a very small hidden constant because the commitments can be represented as bit vectors.

For a constant fraction of corrupted parties, the message complexity is quadratic in n and essentially optimal. We do not know whether the communication complexity can be lowered to a quadratic expression in n as well.

3.4 Atomic Broadcast

Atomic broadcast guarantees a total order on messages such that honest parties deliver all messages with a common tag in the same order. It is well known that protocols for atomic broadcast are considerably more expensive than those for reliable broadcast because even in the crash-fault model, atomic broadcast is equivalent to consensus [16] and cannot be solved by deterministic protocols. The atomic broadcast protocol given here builds directly on multi-valued validated Byzantine agreement from the last section.

3.4.1 Definition

Atomic broadcast ensures that all messages broadcast with the same tag ID are delivered in the same order by honest parties; in this way, ID can be interpreted as the name of a broadcast "channel." The total order of atomic broadcast yields an implicit labeling of all messages. Assuming some honest party has atomically delivered s distinct messages, the global sequence of the first s delivered messages is well-defined. Thus, an explicit sequence number is not needed. Since the sender of a payload message is not necessarily identifiable (without requiring explicit authenticity instead of integrity), the sender name is also omitted, and an unstructured tag ID suffices.

An atomic broadcast is activated when the adversary delivers an input message to P_i of the form

$$(ID, \texttt{in}, \texttt{a-broadcast}, m),$$

where $m \in \{0,1\}^*$. When this occurs, we say *P_i atomically broadcasts m with tag ID*. "Activation" here refers only to the broadcast of a particular payload message; the broadcast channel ID must be opened before the first such request.

A party terminates an atomic broadcast of a particular payload by generating an output message of the form

$$(ID, \texttt{out}, \texttt{a-deliver}, m).$$

In this case, we say *P_i atomically delivers m with tag ID*. To distinguish atomic broadcast from other forms of broadcast, we will also use the terms *a-broadcast* and *a-deliver*.

For the composition of atomic broadcast with other protocols, we need a synchronized output mode, where *a-delivering* a payload may block the protocol and prevent it from delivering more payloads until the consumer is ready to accept them. We introduce an acknowledgment mechanism for output messages for this purpose, i.e., the adversary should *acknowledge* every *a-delivered* payload message to the delivering party. In practice, the *a-delivery* operation could be implemented by a blocking upcall to the higher-level protocol. In terms of the formal model, an acknowledgment is modeled as an input message $(ID, \texttt{in}, \texttt{a-acknowledge})$ from the adversary. When a party receives such a message, it means that its most recently *a-delivered* payload message with tag ID has been *acknowledged*. We will say that the adversary *generates acknowledgments* if it acknowledges every *a-delivered* message.

Again, the adversary must not request an *a-broadcast* of the same payload message from any particular party more than once for each *ID* (however, several parties may *a-broadcast* the same message).

Atomic broadcast protocols should be *fair* so that a payload message m is scheduled and delivered within a reasonable (polynomial) number of steps after it is *a-broadcast* by an honest party. But since the adversary may delay the sender arbitrarily and *a-deliver* an a priori unbounded number of messages among the remaining honest parties, we can only provide such a guarantee when at least $t + 1$ honest parties become "aware" of m. Our definition of fairness requires actually that only after $t + 1$ honest parties have *a-broadcast* some payload, it is guaranteed to be delivered within a reasonable number of steps. It can be interpreted as a termination condition for the broadcast of a particular payload m. This is also the reason for allowing multiple parties to *a-broadcast* the same payload message—a client application might be able to satisfy this precondition through external means and achieve guaranteed fair delivery in this way.

The *efficiency* condition (which ensures fast termination) for atomic broadcast differs from the protocols discussed so far because the protocol for a particular tag cannot terminate on its own. It merely stalls if no more undelivered payload messages are in the system and must be terminated externally. Thus, we cannot define efficiency using the absolute number of protocol messages generated. Instead we measure the progress of the protocol with respect to the number of messages that are *a-delivered* by honest parties. In particular, we require that the number of associated protocol messages does not exceed the number of *a-delivered* payload messages times a polynomial factor, independent of the adversary.

We say that a protocol message is *associated* to the atomic broadcast protocol with tag *ID* if and only if the message is generated by an honest party and tagged with *ID* or with a tag $ID|\ldots$ starting with *ID*. In particular, this encompasses all messages of the atomic broadcast protocol with tag *ID* generated by honest parties and all messages associated to basic broadcast and Byzantine agreement sub-protocols invoked by atomic broadcast.

Fairness and efficiency are defined using the number of payload messages in the "queues" of honest parties. We say that a payload message m is *in the queue of a party* P_i if P_i has *a-broadcast* m with tag *ID*, but not *a-delivered* m tagged with *ID*. The *system queue* contains any message that is in the queue of *some* honest party, but has not yet been *a-delivered* by *any* honest party.

Definition 7 (Atomic Broadcast). A protocol for *atomic broadcast* satisfies the following conditions except with negligible probability:

Validity: If an honest party has *a-broadcast* m tagged with *ID*, then it *a-delivers* m tagged with *ID*, provided the adversary opens channel *ID* for all honest parties, delivers all associated messages, and generates acknowledgments.

Agreement: If some honest party has *a-delivered* m tagged with ID, then all honest parties *a-deliver* m tagged with ID, provided the adversary opens channel ID for all honest parties, delivers all associated messages, and generates acknowledgments for every party that has not yet *a-delivered* m tagged with ID.

Total Order: Suppose an honest party P_i has *a-delivered* $m_1, \dots, m_s$ with tag ID, a distinct honest party P_j has *a-delivered* $m'_1, \dots, m'_{s'}$ with tag ID, and $s \le s'$. Then $m_l = m'_l$ for $1 \le l \le s$.

Integrity: For all ID, every honest party *a-delivers* a payload message m at most once tagged with ID. Moreover, if all parties are honest, then m was previously *a-broadcast* by some party with tag ID.

Fairness: Fix a particular protocol instance with tag ID. For any m consider the system at the time when the $(t+1)$st honest party *a-broadcasts* m with tag ID. Let S_m denote the number of distinct messages *a-delivered* by honest parties up to that time, and let V_m denote the total number of distinct payload messages in the queues of those $t+1$ parties who have *a-broadcast* m (all with tag ID). Suppose the adversary causes some honest party to *a-deliver* m as the W_m-th message. Then the random variable $(W_m - S_m)/V_m$ is polynomially bounded.

Efficiency: For a particular protocol instance with tag ID, let X denote its communication complexity and let Y be the maximum number of distinct payload messages that have been *a-delivered* by some honest party with tag ID. Then the random variable $X/(Y+1)$ is probabilistically polynomially bounded.

Some remarks on the above definition:

1. Validity, agreement, and integrity are analogous to reliable broadcast; only total order and fairness are new. Validity ensures liveness of a protocol and rules out trivially empty protocols. It is stated in the canonical form (only the sender should *a-deliver* the message).

2. The agreement condition combines the consistency and totality of reliable broadcast; there is no need to distinguish these two aspects here, but they could also be separated. In particular, only totality requires that messages and acknowledgments are delivered.

3. In the fairness condition, note that $V_m \ge 1$ by definition. One could define a weaker version of fairness and start counting only if f honest parties *a-broadcast* a request for $t+1 \le f \le n-t$.

4. The efficiency condition counts only the payload messages delivered by the "fastest" honest party. This party will usually be synchronized within one round with at least $n - 2t - 1$ other honest parties, but it seems impossible to synchronize it with the "slowest" honest party. Moreover, there seems to be no easy way to provide a fixed bound on a suitable statistic (such as communication complexity) until *all* honest parties have delivered a particular payload. This is because the adversary can always drive forward system with only $n - 2t$ honest parties and leave the others behind. The "fast" parties might generate an a priori unbounded amount of work until the "slow" ones finally *a-deliver* a particular payload, if at all. (Adding 1 to the divisor covers the state until the first payload is delivered.)

3.4.2 A Protocol for Atomic Broadcast

We now present a protocol for atomic broadcast based on validated Byzantine agreement. Its overall structure is similar to the protocol of Hadzilacos and Toueg [37] for the crash-fault model, but we need to take additional measures to tolerate Byzantine faults.

Our Protocol ABC for atomic broadcast proceeds as follows. Each party maintains a FIFO queue of not yet *a-delivered* payload messages. Messages received to *a-broadcast* are appended to this queue whenever they are received. The protocol proceeds in asynchronous global rounds, where each round r consists of the following steps:

1. Send the first payload message w in the current queue to all parties, accompanied by a digital signature σ in an `a-queue` message.
2. Collect the messages of $n - t$ distinct parties and store them in a vector W, store the corresponding signatures in a vector S, and propose W for Byzantine agreement validated by S.
3. Perform multi-valued Byzantine agreement with validation of a vector $W = [w_1, \ldots, w_n]$ and proof $S = [\sigma_1, \ldots, \sigma_n]$ through the predicate $Q_{ID|\mathsf{abc}|r}(W, S)$ which is true if and only if for at least $n - t$ distinct indices j, the vector element σ_j is a valid $\mathcal{S}$-signature on $(ID, \texttt{a-queue}, r, j, w_j)$ by P_j.
4. After deciding on a vector V of messages, deliver the union of all payload messages in V according to a deterministic order; proceed to the next round.

In order to ensure liveness of the protocol, there are actually two ways in which the parties move forward to the next round: when a party receives an *a-broadcast* input message (as stated above) and when a party receives an `a-queue` message of another party pertaining to the current round. If either of these two messages arrive and contain a yet undelivered payload message, and if the party has not yet sent its own `a-queue` message for the current round, then it enters the round by appending the payload to its queue and sending an `a-queue` message to all parties.

The detailed description of Protocol ABC is found in Figure 5. The FIFO queue q is an ordered list of values (initially empty). It is accessed using the operations *append*, *remove*, and *first*, where *append*(q, m) inserts m into q at the end, *remove*(q, m) removes m from q (if present), and *first*(q) returns the first element in q. The operation $m \in q$ tests if an element m is contained in q.

A party waiting at the beginning of a round simultaneously **waits for** `a-broadcast` and `a-queue` messages containing some $w \notin d$ in line 2. If it receives an *a-broadcast* request, the payload m is appended to q. If only a suitable `a-queue` protocol message is received, the party makes w its own message for the round, but does not append it to q. It should be clear from the protocol that no honest party is ever blocked waiting for some payload message to process if some honest party has *a-broadcast* one and all associated messages have been delivered.

The term $n - t$ in line 9 of the protocol and in the validation predicate $Q_{ID|\mathsf{abc}|r}$ could be replaced by any f' between $t + 1$ and $n - t$ if the fairness condition is changed such that $f = n - f' + 1$ parties instead of $t + 1$ must have *a-broadcast* the message.

The protocol in Figure 5 is formulated using a single loop that runs forever after initialization; this is merely for syntactic convenience and can be implemented by decomposing the loop into the respective message handlers.

Theorem 12. *Given a protocol for multi-valued validated Byzantine agreement and assuming $\mathcal{S}$ is a secure signature scheme, Protocol* ABC *provides atomic broadcast for $n > 3t$.*

Proof. We first prove *validity.* Towards a contradiction, suppose that some honest party has *a-broadcast* a payload message m, but not *a-delivered* it and yet, all associated protocol messages and acknowledgments have been delivered. Since the sender has *a-broadcast* but not *a-delivered* m, its queue contains at least m and it can no longer be waiting in line 2. Thus, it has proceeded and sent `a-queue` messages to all parties in line 8. Since these have been delivered, every honest party has received an `a-queue` message containing $m \notin d$ and therefore has also entered the same round (by condition for waiting in line 2). Thus, all $n - t$ honest parties have sent valid `a-queue` messages and every honest party has received all of them and subsequently started and terminated Byzantine agreement. Since also the *a-delivered* payloads have been acknowledged, the sender must be waiting in line 2 with $q = []$. But then m has been removed from q and this occurs only if it was *a-delivered*, a contradiction.

We now establish *agreement.* Towards a contradiction, suppose that some honest

Protocol ABC **for party** P_i **and tag** ID

LET $Q_{ID|\mathtt{abc}|r}$ BE THE FOLLOWING PREDICATE:

$$Q_{ID|\mathtt{abc}|r}([w_1,\ldots,w_n],[\sigma_1,\ldots,\sigma_n]) \equiv \big(\text{for at least } n-t \text{ distinct } j,\ \sigma_j \text{ is a valid } \mathcal{S}\text{-signature by } P_j \text{ on } (ID,\mathtt{a\text{-}queue},r,j,w_j).\big)$$

INITIALIZATION:

```
    $q \leftarrow []$        {FIFO queue of messages to *a-broadcast*}
    $d \leftarrow \emptyset$        {set of *a-delivered* messages}
    $r \leftarrow 0$        {current round}
```

UPON RECEIVING MESSAGE $(ID, \mathtt{in}, \mathtt{a\text{-}broadcast}, m)$:

```
    if $m \notin d$ and $m \notin q$ then
        $append(q, m)$
```

FOREVER:

```
 1: $w_j \leftarrow \perp; \sigma_j \leftarrow \perp$    $(1 \le j \le n)$
 2: wait for $q \neq []$ or a message $(\mathtt{a\text{-}queue}, r, l, w_l, \sigma_l)$ received from $P_l$
        such that $w_l \notin d$ and $\sigma_l$ is a valid signature from $P_l$
 3: if $q \neq []$ then
 4:     $w \leftarrow first(q)$
 5: else
 6:     $w \leftarrow w_l$
 7: compute a digital signature $\sigma$ on $(ID, \mathtt{a\text{-}queue}, r, i, w)$
 8: send the message $(\mathtt{a\text{-}queue}, r, i, w, \sigma)$ to all parties
 9: wait for $n-t$ messages $(\mathtt{a\text{-}queue}, r, j, w_j, \sigma_j)$ such that $\sigma_j$ is a valid
        signature from $P_j$ (including the message from $P_l$ above)
10: $W \leftarrow [w_1, \ldots, w_n]; S \leftarrow [\sigma_1, \ldots, \sigma_n]$
11: propose $W$ validated by $S$ for multi-valued validated Byzantine agreement
        on $ID|\mathtt{abc}|r$ with predicate $Q_{ID|\mathtt{abc}|r}$
12: wait for the validated Byzantine agreement protocol to decide some
        $V = [v_1, \ldots, v_n]$ for $ID|\mathtt{abc}|r$
13: $b \leftarrow \bigcup_{j=1}^{n} v_j$
14: for $m \in (b \setminus d)$, in some deterministic order do
15:     output $(ID, \mathtt{out}, \mathtt{a\text{-}deliver}, m)$
16:     wait for an acknowledgment
17:     $d \leftarrow d \cup \{m\}$
18:     $remove(q, m)$
19: $r \leftarrow r + 1$
```

Figure 5: Protocol ABC for atomic broadcast using multi-valued validated Byzantine agreement.

P_i has *a-delivered* a payload message m, but an honest P_j has not *a-delivered* it and yet, all associated protocol messages have been delivered and acknowledgments have been generated for all parties who have not yet *a-delivered* m. Assume P_i *a-delivered* m in round r. Since no party who has not *a-delivered* m is blocked waiting for messages or acknowledgments under these conditions, it is easy to see from inspection of the protocol and from the liveness condition of the Byzantine agreement sub-protocol that P_j must have received all messages belonging to any round up to and including r. But then it cannot be waiting for an acknowledgment either—unless it has already *a-delivered* m.

The *total order* condition follows from the agreement property of the validated Byzantine agreement primitive since all honest parties decide on the same proposal and then *a-deliver* all payload messages contained in the proposal in a deterministic order. This implies also that the set d of *a-delivered* messages is the same for all honest parties.

Integrity is immediate from the protocol by induction on the construction of d, using the properties of Byzantine agreement. Even if corrupted parties include messages that have already been delivered, they are not delivered again.

To show *fairness*, consider the system when W_m has been defined. We have to provide a polynomial bound on $(W_m - S_m)/V_m$, independent of the adversary. Note that the decided vector of payloads in every round contains $n - t$ values of which at least $n - 2t \geq t + 1$ are the first elements in the queues of honest parties. Thus, at least one of the initially V_m elements in the respective $t+1$ queues has been *a-delivered* in every round henceforth. But there are at most n distinct payloads that are *a-delivered* per round, which implies that $W_m - S_m$ is bounded by nV_m.

For *efficiency*, we have to relate the communication complexity of the protocol to the payload messages that were actually *a-delivered*. Note that honest parties generate messages only when they make progress in the round structure—either by sending an `a-queue` message or by invoking the Byzantine agreement sub-protocol. But an honest party enters the next round only if it is aware of some payload message that it has not yet *a-delivered*. Since at least one payload message from the system queue is delivered in every round, all protocol messages generated during that round can be related to that payload. There are a fixed polynomial number of protocol messages generated directly by the protocol in every round and the length of each one is at most n times the length of a payload. The communication complexity of the Byzantine agreement sub-protocol is probabilistically polynomially bounded by its efficiency condition. Thus, the communication complexity per round is probabilistically polynomially bounded. □

The message complexity of Protocol ABC to broadcast one payload message m is dominated by the number of messages in the multi-valued validated Byzantine agreement; the extra overhead for atomic broadcast is only $O(n^2)$ messages. The same holds for the communication complexity, but the proposed values have length $O(n(|m|+K))$, assuming digital signatures of length K bits.

With Protocol VBAconst from Section 3.3.4, the total expected message complexity is $O(n^2)$ and the expected communication complexity is $O(n^3|m|)$ for an atomic broadcast of a single payload message.

3.4.3 Equivalence of Byzantine Agreement and Atomic Broadcast

For the sake of completeness, we state the equivalence of atomic broadcast to Byzantine agreement in the cryptographic model. It is the analogue to the equivalence between consensus and atomic broadcast in the crash-fault model shown by Chandra and Toueg [16].

Corollary 13. *(Binary) Byzantine agreement and atomic broadcast are equivalent in the basic system model of Section 3.1.1, assuming a secure signature scheme and provided $n > 3t$.*

Proof. To implement Byzantine agreement from an atomic broadcast protocol, a party uses the following algorithm:

1. To propose $v \in \{0,1\}$ for transaction ID, compute a digital signature σ on (ID, v) and *a-broadcast* the message (ID, v, σ).
2. Wait for *a-delivery* of the first $2t+1$ messages of the form (ID, v_j, σ_j) from distinct parties that contain valid signatures. Decide for the simple majority of all received values v_j.

The other direction follows from Theorems 8 and 12. □

Note that using an appropriately defined notion of *authenticated* atomic broadcast, this could also be implemented without the additional digital signatures in the reduction. However, Protocol ABC would have to be modified in order to provide authentication.

3.5 Secure Causal Atomic Broadcast

Secure causal atomic broadcast (SC-ABC) is a useful protocol for building secure applications that use state machine replication in a Byzantine setting. It provides atomic broadcast, which ensures that all recipients receive the same sequence of messages, and also guarantees that the payload messages arrive in an order that maintains "input causality," a notion introduced by Reiter and Birman [56]. Informally, input causality ensures that a Byzantine adversary may not ask the system to deliver any payload message that depends in a meaningful way on a yet undelivered payload sent by an honest client. This is very useful for delivering client requests to a distributed service in applications that require the contents of a request to remain secret until the system processes it. Input causality is related to the standard causal order (going back to Lamport [40]), which is a useful safety property for distributed systems with crash failures, but is actually not well defined in the Byzantine model [37].

Input causality can be achieved if the sender encrypts a message to broadcast with the public key of a threshold cryptosystem for which all parties share the decryption key [56]. The ciphertext is then broadcast using an atomic broadcast protocol; after delivering it, all parties engage in an additional round to recover the message from the ciphertext.

In our description of secure causal atomic broadcast, one of the parties acts as the sender of a payload message. If SC-ABC is used by a distributed system to broadcast client requests, then encryption and broadcasting is taken care of by the client. In this case, additional considerations are needed to ensure proper delivery of the replies from the service (see [56] for those details).

3.5.1 Definition

Associated with any instance of a secure causal atomic broadcast protocol with tag ID is an encryption algorithm E_{ID}. It should be possible to infer this algorithm from the dealer's public output. E_{ID} is a probabilistic algorithm that maps a message m to a ciphertext c. We call $c = E_{ID}(m)$ an encryption of m (with tag ID). Since the encryption algorithm is probabilistic, there will in general be many different encryptions of a given message; indeed, this will necessarily be the case if the system is to be secure.

An application that wants to securely broadcast a payload message should first encrypt it using E_{ID} and invoke the broadcast protocol with the resulting ciphertext. Since E_{ID} is publicly known, also clients from outside the group $P_1, \ldots, P_n$ can produce ciphertexts.

A secure causal atomic broadcast protocol is activated when P_i receives an input message of the form

$$(ID, \texttt{in}, \texttt{s-broadcast}, c).$$

We say P_i *s-broadcasts* c with tag ID.

Unlike atomic broadcast, delivery consists of two distinct steps: the first is the generation of an output message of the form

$$(ID, \texttt{out}, \texttt{s-schedule}, c),$$

and the second is the generation of an output message of the form

$$(ID, \texttt{out}, \texttt{s-reveal}, m).$$

We shall require that honest parties generate sequences of such pairs of output messages—there must never be two consecutive `s-schedule` or `s-reveal` messages. When the `s-schedule` message is generated, we will say that P_i *s-schedules* the ciphertext c (with tag ID). When the `s-reveal` message is generated, we will say that P_i *s-delivers* the ciphertext c (with tag ID), where c is the most recently *s-scheduled* ciphertext; we call m the *associated cleartext.*

Definition 8 (Secure Causal Atomic Broadcast). A secure causal atomic broadcast protocol satisfies the properties of an atomic broadcast protocol, where the *s-broadcast* and *s-delivery* of ciphertexts in the secure causal atomic broadcast protocol play the role of the *a-broadcast* and *a-delivery* of payload messages in an atomic broadcast protocol.

Additionally, the following conditions hold.

Serialization: Honest parties strictly alternate between generating `s-ready` and `s-deliver` output messages with the same tag. Moreover, if some honest party outputs an `s-ready` message containing a bit string c, then the next `s-deliver` output contains a pair (m, c) where m results from the threshold-decryption of a ciphertext identified by c.

Message Secrecy: According to the basic system model, the parties run an atomic broadcast protocol (and possibly other broadcast protocols), and the adversary plays the following game:

B1. The adversary interacts with the honest parties in an arbitrary way.

B2. The adversary chooses two messages m_0 and m_1 and a tag ID; it gives them to an "encryption oracle." The oracle chooses a bit B at random and computes an encryption c of m_B with tag ID, and gives this ciphertext to the adversary.

B3. The adversary continues to interact with the honest parties subject only to the condition that no honest party *s-schedules* c with tag ID.

B4. Finally, the adversary outputs a bit $\hat{B}$.

Then, for any adversary the probability that $\hat{B} = B$ must exceed $\frac{1}{2}$ only by a negligible amount.

Message Integrity: According to the basic system model, the parties run an atomic broadcast protocol (and possibly other broadcast protocols), and the adversary plays the following game:

C1. The adversary interacts with the honest parties in an arbitrary way.

C2. The adversary chooses a message m and a tag ID, and gives it to an "encryption oracle." The oracle computes an encryption c of m with tag ID, and gives this ciphertext to the adversary.

C3. The adversary continues to interact with the honest parties in an arbitrary way.

We say the adversary wins the game if at some point an honest party *s-delivers* c with tag ID, but corresponding cleartext m' is not equal to m. Then, for any adversary, the probability that it wins this game is negligible.

Message Consistency: If two parties honest parties *s-deliver* the same ciphertext c with tag ID, then with all but negligible probability, the associated cleartexts are the same.

It is easy to verify that this definition implies input causality in the sense of Reiter and Birman [56], i.e., that a cleartext remains hidden from the adversary until the corresponding ciphertext is *s-scheduled*. But the cleartext may be revealed to the adversary before the first honest party outputs it in a `s-reveal` message, and this is also the reason for introducing our two-step delivery process. Although this is necessary for the proper definition of security, *s-scheduling* a ciphertext might be omitted in a practical implementation.

The *message integrity* condition gives clients access to the broadcast protocol for cleartext payload messages, and implies that payloads contained in correctly encrypted ciphertexts are actually output by the honest parties.

3.5.2 A Protocol for Secure Causal Atomic Broadcast

Protocol SC-ABC in Figure 6 implements secure causal atomic broadcast. It uses an $(n, t+1)$-threshold cryptosystem $\mathcal{E}_1$ that is secure against adaptive chosen ciphertext attacks (see Section 3.1.3.3) for which the parties share the decryption key. It also uses an atomic broadcast protocol according to Section 3.4.

During initialization, the dealer generates a public key for $\mathcal{E}_1$, together with the corresponding private key shares, and distributes them according to the initialization algorithm of $\mathcal{E}_1$.

For a tag ID, $E_{ID}(m)$ is computed by applying the encryption algorithm of $\mathcal{E}_1$ to m with label ID, using the generated public key of the cryptosystem.

We emphasize that all instances of the secure causal broadcast protocol share the same public key for $\mathcal{E}_1$, and so the use of *labeled ciphertexts* is essential to properly "isolate" different instances of the protocol from one another.

To *s-broadcast* a ciphertext c, we simply *a-broadcast* c. Upon *a-delivery* of a ciphertext c, a party *s-schedules* c. Then it computes a decryption share δ and sends this to all other parties in an `s-decrypt` message containing c. It waits for $t+1$ `s-decrypt` messages pertaining to c. Once they arrive, it recovers the associated cleartext and *s-delivers* c. After receiving the acknowledgment, the party continues processing the next *a-delivery* by generating the corresponding acknowledgment. The details are in Figure 6. For ease of notation, the protocol in Figure 6 is formulated using a FOREVER loop; it can be decomposed into the respective message handlers in straightforward way.

```
Protocol SC-ABC for party $P_i$ and tag $ID$

INITIALIZATION:
    open an atomic broadcast channel with tag $ID|\texttt{scabc}$

UPON RECEIVING $(ID, \texttt{in}, \texttt{s-broadcast}, c)$:
    a-broadcast $c$ with tag $ID|\texttt{scabc}$

FOREVER:
    wait for the next message $c$ that is a-delivered with tag $ID|\texttt{scabc}$
    compute an $\mathcal{E}_1$-decryption share $\delta$ for $c$ with label $ID$
    output $(ID, \texttt{out}, \texttt{s-schedule}, c)$
    send the message $(\texttt{s-decrypt}, c, \delta)$ to all parties
    $\delta_j \leftarrow \perp \qquad (1 \leq j \leq n)$
    wait for $t+1$ messages $(\texttt{s-decrypt}, c, \delta_j)$ from distinct parties that contain valid
      decryption shares for $c$ with label $ID$
    combine the decryption shares $\delta_1, \ldots, \delta_n$ to obtain a cleartext $m$
    output $(ID, \texttt{out}, \texttt{s-reveal}, m)$
    wait for an acknowledgment
    acknowledge the last a-delivered message with tag $ID|\texttt{scabc}$
```

Figure 6: Protocol SC-ABC for secure causal atomic broadcast.

Theorem 14. *Given an atomic broadcast protocol and assuming $\mathcal{E}_1$ is a $(n, t+1)$-threshold cryptosystem secure against adaptive chosen-ciphertext attacks, Protocol* SC-ABC *provides secure causal atomic broadcast for $n > 3t$.*

Proof. We have to show that the protocol implements atomic broadcast and satisfies message secrecy and message integrity conditions.

We first show *validity.* Suppose an honest party P_i has *s-broadcast* c and all associated messages have been delivered and all acknowledgments have been generated. Thus, P_i has *a-broadcast* c. By the validity of the atomic broadcast protocol and because the messages associated to the secure broadcast contain also all those associated to the atomic broadcast, c has been *a-delivered.* According to the agreement condition of atomic broadcast, all honest parties have therefore generated decryption shares for c and sent `s-decrypt` messages to all parties. Thus, P_i has received at least $t + 1$ valid shares for c. But then P_i has also *s-delivered* c.

It is perhaps interesting to note that the above proof of *validity* made essential use of *both* the *validity* and *agreement* properties of the underlying atomic broadcast protocol.

For *agreement*, suppose that an honest P_i has *s-delivered* c and P_j has not, and yet, all associated messages have been delivered and acknowledgments have been generated for those parties who have not *s-delivered* c. By the agreement condition of the underlying atomic broadcast, all other honest parties must also have *a-delivered* c. Thus, they have generated decryption shares and also P_j has received at least $t + 1$ valid shares for c. Therefore, P_j must have *s-delivered* c, a contradiction.

To show *efficiency*, we must bound the amount of work done (as measured by communication complexity) per *s-delivered* message. But since the *s-delivery* messages is synchronized with the *a-delivery* of ciphertexts in Protocol SC-ABC, the number of *a-delivered* messages exceeds the number of *s-delivered* ones by at most one, and efficiency follows from the efficiency condition of the atomic broadcast protocol.

Note that without this synchronization, we could not achieve efficiency, since the lower level atomic broadcast protocol could "run ahead" of the higher level secure causal atomic broadcast protocol—lots of messages would be generated, but very few messages would be *s-delivered.*

It is easy to see that the remaining broadcast properties (*total order*, *integrity*, and *fairness*) hold as well, using the corresponding properties of the underlying atomic broadcast.

Message secrecy, integrity, and consistency follow easily from the properties of the underlying threshold encryption scheme. □

4 Applications

Our distributed trusted services are based on secure state machine replication in the Byzantine model. Requests to the services are delivered by the broadcast protocols mentioned in the previous chapter. A broadcast is started when the client sends a message containing the request to a sufficient number of servers. In general, the client must send the request to more than t servers or a corrupt server could simply ignore the message; alternatively, one could postulate that one server acts as a gateway to relay the request to all servers and leave it to the client to resend its message if it receives no answer within the expected time.

Depending on whether it needs to maintain causality among client requests, a service may use atomic broadcast directly or secure causal atomic broadcast otherwise. If the client requests commute, reliable broadcast suffices.

Each server returns a partial answer to the client, who must wait for at least $2t+1$ values before determining the proper answer by majority vote. Since atomic broadcast guarantees that all servers process the same sequence of requests, the client will obtain the same answer from all honest servers. If the application returns a digital signature, the answers may contain signature shares from which the client can recover a threshold signature.

Four applications of our protocols to distributed trusted third-party services are described here: a digital notary, a trusted party for optimistic fair exchange, a certification authority, and a basic authentication server.

4.1 Digital Notary Services

One of the simplest state machines is a single counter. Despite this simplicity, there are a number of applications in which a single counter provided by a trusted central authority is of fundamental importance. We will describe in moderate detail one such service—a *digital notary service*—and then briefly mention others.

In its most basic form, a digital notary service receives documents, assigns a sequence number to them and certifies this by its signature. In order to focus on the needed security and verification features of such a service, it is convenient to think of the example of a patent office: individuals or organizations in the research community create new intellectual property and wish to protect that property by applying to the patent office. This office must in turn assign each application a sequence number—essentially a logical timestamp—so that as the review process takes place, an application which treads too close to one with an earlier acceptance number can be denied, while the earlier one is given priority and that applicant owns the resulting patent.

When the patent office is a single physical location, the applications can simply be given consecutive numbers as they come in the front door and the community at large must simply trust that what happens inside that building is honest and reliable; something not much different can also be done when the office moves to a single server on the Internet. It is, however, wise to distribute such a service among $n > 1$ hosts running different software under separate system administration and in separate physical locations, in order to remove the single point of failure and to provide a more robust service. Another situation in which such a distributed patent office might arise would be if several nations agreed to cooperate in the issuing of patents, so that all applications would be made simultaneously to all national offices, which would all agree on an ordering of these submissions.

As we have argued elsewhere, a reasonable model then to apply for the distributed servers is one of processes communicating asynchronously over a network which is presumed to be in the hands of the adversary, who is also presumed to have corrupted up to $t < n/3$ of the servers. Several security concerns must be satisfied if the distributed patent office is to continue to function:

- the documents must be processed *atomically*, i.e., despite asynchronous communication, each document must be given a unique sequence number, and all honest hosts must agree on the number assigned to a given application;
- documents must be processed *securely*, in that the contents of a document should not be readable by corrupt servers until the protocols have finished and the document have been given their sequence number—otherwise the adversary in the patent application example could insert an application for a similar invention into the sequencing protocol ahead of the legitimate applicant;
- the resulting sequence numbers should be *signed* in such a way that the client can be confident that her number is provably legitimate, and not merely the result of deceptive messages from the adversary.

It should then be clear that the essential tool to realize this task in a reliable and efficient manner is the *secure causal atomic broadcast* (SC-ABC) of Chapter 3, in its parallel activation mode, with a minor addition to produce the desired signatures. In particular:

1. The dealer should create keys for an $(n, t+1, t)$-threshold signature scheme $\mathcal{S}_2$ and give them to the notary servers, and the initialization phase of each server should zero a counter *seqnum*.

2. Notary servers wait for the *s-delivery* of a message with tag `notarize`, whose payload m contains the document to be notarized.

3. When such a message is *s-delivered*, each server should increment the *seqnum* counter, create an $\mathcal{S}_2$-signature share σ on the message (m, seqnum) and transmit $(\mathit{seqnum}, \sigma)$ to the original client.

The client uses this service by

1. computing an $\mathcal{E}_1$-encryption c of its document m containing also information which names the client and her internal reference number for this notary request,

2. *s-parallel-broadcasting* payload c with tag `notarize` to all notary servers,

3. waiting for $t + 1$ distinct servers to reply with a sequence number and valid $\mathcal{S}_2$-signature share and then

4. assembling the shares into a $\mathcal{S}_2$-signature on the message (m, seqnum).

One inelegant aspect of the notary service just described is that it seems to put some of the onus of dealing with the distributed service on the individual clients, in that the client must herself assemble the signature shares at the end of the process. This could be avoided, at the cost of an additional round of communication among the servers, by having each one broadcast its $\mathcal{S}_2$-signature share to its peers and then wait for enough such shares to come back from other servers so that it could assemble the signature itself. However, the client's initial step of encrypting her message before broadcasting it to all servers cannot be avoided, as she cannot trust any other party with the cleartext of her submission until the distributed notary service has issued a signed sequence number for that request. In practice, the encryption of the client's outgoing message and the assembly of a signature from the incoming server messages would be handled by routines in the communications software library, so these internals would not be the concern of the client application at all.

Several other distributed services with a similar need of a synchronized and signed sequence number created by processing, until the last step, an encrypted form of the client request can all fall under the rubric of a *digital bidding service*. Here we are imagining a valuable item for which the order of arrival of the bids as well as the details of the bids themselves are used to allocate the good or goods to (some of) the clients. Stocks, where both the order in which the offers are made as well as the offered purchase price, or government contracts, where again the priority of the bid as well its specific terms are used to assign the contract, are both items which would benefit from such a service. In individual applications, the details of the small programs which use the SC-ABC protocol may vary, such as particular requirements of client authentication or creation of a mechanism to terminate a bidding process, but the general outline would be very similar to the above example. (For a discussion of some of these and related issues and examples, see [56].)

4.2 Fair Exchange TTPs

Commercial interactions between two actors over the Internet must all deal, in some way, with the *fair exchange problem*: how the participants can exchange two valuable tokens in such a way that either both get the item they bargained for or neither does. Many protocols have appeared in the literature to solve this problem, and they all use the mechanism of a trusted third party in some way (at least all potentially practical protocols do so). Perhaps the most efficient algorithms are those which go under the name of *optimistic fair exchange*, where the third party is only involved when the transaction fails, either to abort a transfer when the initiating party is not releasing her valuable item, or to force a conclusion of the transaction if the first party has released her good but the second is trying to avoid the promised payment—or simply if some of the protocol messages are lost or deleted by a malicious network.

Several very nice algorithms for optimistic fair exchange are given by Asokan, Shoup and Waidner in [2] for the exchange of digital signatures. They have the advantage of being extremely flexible, so they can operate on all commonly used signature schemes and can be easily adapted for the exchange of digital content or certified e-mail, for example. It is very convenient for our current setting that the model in [2] is the same as our present one, with asynchronous communication on an untrusted network. We shall describe here the special case from [2] of a protocol for the electronic signing of contracts, because it is one that can very easily take advantage of the communication primitives we have developed.

Let us denote by $[\alpha]_X$ the bit string α concatenated with a signature on α under X's public key. Then the protocol for optimistic fair exchange of digital signatures on a contract m between two parties Alice and Bob, with dispute resolution by Tom, proceeds as follows (we use A as an abbreviation for "Alice" where convenient, and likewise with B and T):

1. Alice sends Bob $[m, A, B, T]_A$.
2. Bob receives and verifies this signature, replying with $[m, A, B, T]_B$ if successful and quitting otherwise.
3. Alice receives and verifies this reply, sending him $\sigma_A = [m, A, B]_A$ if successful and requesting an *abort* from Tom if not.

4. Bob receives σ_A from Alice, sending her $\sigma_B = [m, A, B]_B$ if he was able to verify σ_A or else requesting *resolve* from Tom.

5. Alice receives and checks this σ_B from Bob, outputting the (σ_A, σ_B) if satisfied or otherwise requesting Tom to *resolve* the exchange.

In this scheme, a *valid contract* is a string of the form

$$([m, A, B]_A, [m, A, B]_B)$$

or, in the case that Tom has had to intervene,

$$[[m, A, B, T]_A, [m, A, B, T]_B]_T;$$

the latter is called a *proxy signature*, and we are assuming that the social infrastructure is in place to enforce it legally as completely as a normal contract.

There are two requests the trustworthy Tom must be able to handle: an *abort* from Alice and *resolves* from Alice or Bob. The *abort* is essentially a request from Alice that all future *resolves* from Bob on the contract m be disallowed. If, however, Bob has already *resolved*, Tom can and does directly deliver the proxy signature to Alice.

Either Alice or Bob may attempt to *resolve* by sending Tom the message

$$m_{resolve} = ([m, A, B, T]_A, [m, A, B, T,]_B),$$

to which Tom replies with $[m_{resolve}]_T$ if no abort has yet been processed.

In this optimistic protocol, it is expected that Alice and Bob will only turn to the TTP for conflict resolution—in which case Tom must always be able to respond reliably. To increase the robustness of this TTP service, it is again natural to distribute it among several hosts on the Internet with, as usual, as varied a configuration as possible. In the course of normal, friendly interactions, Alice and Bob will not go to the TTP at all, and hence there need be no change whatsoever in the above protocol for their personal communication.

What must change for a distributed TTP is the handling of the *abort* and *resolve* sub-protocols. In particular, Tom has some state information which must be maintained in a consistent fashion across the separate servers. Also, the order of processing of requests must be the same at all instances of the TTP, since *abort* and *resolve* do not commute. These basic consistency requirements are amply met by the *atomic broadcast* (ABC) primitive described in Chapter 3, whose *validity*, *agreement*, *integrity* and *total order* properties are designed exactly for this kind of state machine replication problem.

A possible alternative is to use Chapter 3's *secure causal atomic broadcast* (SC-ABC), which differs from ABC mostly in that it keeps communications regarding a certain request encrypted until that request is committed, i.e., *s-delivered.* This can be useful in our present context because we are explicitly assuming, in order for our new model to be more robust than the single TTP scenario, that some of the TTP servers are corrupt. While the dishonest TTP servers cannot forge a proxy contract or abort receipt, since these require a threshold signature which they do not have sufficient shares to assemble, they can influence the outcome of the ordering negotiations which make up the ABC protocol, and decide to do so based on whether the requests are for *abort* or *resolve.* Since it is our philosophy that the adversary should not be able to squeeze any possible advantage in any imaginable situation from peeking inside the secure envelope of our system communications (so that, for example, we use only cryptographic primitives which are secure against chosen-ciphertext attacks), we will present here the version of distributed fair exchange protocols which use the more secure SC-ABC channel, despite it's increased overhead.

Therefore Alice and Bob must issue their various *abort* or *resolve* requests to the distributed TTP via the pre-encrypted *s-parallel-broadcast* channel, as follows. Let ℓ be a bit string agreed upon by Alice and Bob to uniquely identify their transaction (such as a hash of the contract itself, $\ell = H(m)$) and assume that all the TTP servers have key shares for an $(n, t+1, t)$-threshold signature scheme $\mathcal{S}_2$ as well as a state variable S_ℓ for each such ℓ.

Assuming that Alice, Bob and the TTP servers $\{\mathrm{Tom}_i\}_{i=1}^n$ have opened a secure causal atomic broadcast channel with tag `fair-exchange`, here are (some of) the details of the TTP sub-protocols:

Sub-protocol *abort*:

1. Alice computes an $\mathcal{E}_1$-encryption c of ℓ and the string $m_{abort} = [m, A, B, \mathtt{abort}]_A$ and transmits the message

$$(\mathtt{fair\text{-}exchange}, \mathtt{s\text{-}parallel\text{-}broadcast}, c)$$

   to all TTP servers.

2. Each TTP server Tom_i runs the SC-ABC protocol until it achieves the *s-delivery* of a message containing ℓ and m_{abort} with tag `fair-exchange`.

3. Tom_i verifies Alice's signature in m_{abort}, terminating if it is invalid.

4. He next checks if his S_ℓ contains a string beginning with an $m_{resolve}$. If not, he generates an $\mathcal{S}_2$-signature share s_{abort} on m_{abort} and sets $S_\ell = (m_{abort}, s_{abort})$.

5. Finally, he sends whatever is in his S_ℓ to Alice.

6. Alice waits for $2t+1$ servers to reply with a message and valid $\mathcal{S}_2$-signature share and sees which message, an m_{abort} or an $m_{resolve}$, is the majority response.

7. She assembles the signature shares on that majority and now has either a TTP-signed receipt for her *abort* request or a TTP-signed proxy contract which she can try to enforce.

Sub-protocol *resolve* for party X =Alice **or** Bob**:**

1. X computes an $\mathcal{E}_1$-encryption c containing ℓ and the string

$$m_{resolve} = ([m, A, B, T]_A, [m, A, B, T]_B)$$

and transmits the message

$$(\texttt{s-parallel-broadcast}, c)$$

with tag `fair-exchange` to all TTP servers.

2. Each TTP server Tom$_i$ runs the SC-ABC protocol until it achieves the *s-delivery* of a message containing ℓ and $m_{resolve}$ with tag `fair-exchange`.

3. Tom$_i$ verifies both Alice and Bob's signatures in $m_{resolve}$, terminating if either is invalid.

4. He next checks if his S_ℓ contains a string beginning with an m_{abort}. If not, he generates an $\mathcal{S}_2$-signature share $s_{resolve}$ on $m_{resolve}$ and sets $S_\ell = (m_{resolve}, s_{resolve})$.

5. Finally, he sends whatever is in his S_ℓ to X.

6. X waits for $2t + 1$ servers to reply with a message and valid $\mathcal{S}_2$-signature share and sees which message, an $m_{resolve}$ or an m_{abort}, is the majority response.

7. He or she then assembles the signature shares on that majority and now has either a TTP-validated proof that the transaction was aborted or a TTP-signed proxy contract.

The description we have given here appears to be synchronous, but this is merely an artifact of the way we have presented it (for clarity). In fact the actions of the participants Alice and Bob do proceed in a linear fashion as explained above. The various Tom$_i$'s instead spend most of their time running the SC-ABC protocol and then reply to the appropriate exchange partner with the appropriate message, based on their state and whichever message was finally *s-delivered*, an *abort* or *resolve*.

Finally, let us mention that Asokan, Shoup and Waidner point out a nice feature of the above fair exchange protocol is that the TTP can be held accountable for its actions; that is, a dishonest TTP can have its perfidy exposed (essentially by using the signed *abort* receipt). In our model, we are already assuming that t of the Tom_i's are corrupt, but we are also requiring that $n - t$ are honest. Hence there are always enough honest TTP servers around to provide a sufficient number of threshold signature shares and in fact it is unnecessary ever to hold accountable a corrupt server Tom_i, although in practice it may be useful to notice when a particular TTP server has proven itself to be corrupt.

4.3 Certification Authority and Directory Service

A *certification authority* (CA) is a service run by a trusted organization that verifies and confirms the validity of public keys. The issued *certificates* usually also confirm that the real-world user defined in a certificate is in control of the corresponding private key. A certificate is simply a digital signature under the CA's private signing key on the public key and the identity (ID) claimed by the user.

The CA has published its own public key of a digital signature scheme. When a user wants to obtain a certificate for his public key, he sends it together with his ID and credentials to the CA. The ID might consist of name, address, email, date of birth, and other data to uniquely identify the holder. Then the CA verifies the credentials, produces a certificate if they pass, and sends the answer back to the user. The user can verify his certificate with the public key of the CA. For its certificates to be meaningful, the CA must have a clearly stated and publicized policy that it follows for validating public keys and IDs; this policy might change over time.

A *secure directory* service maintains a database of entries, processes lookup queries, and returns the answers authenticated by a signature under its private signing key. The corresponding signature verification key is available to all clients. Several examples of secure directories exist in distributed systems today and more are need in the future, like authentication for the Internet's domain name system [25].

Internally, a secure directory works much like a CA: It receives a query, retrieves some values from the stored database, generates a digital signature on the result, and sends both back to the client. Additional functionality is needed for updating the database.

Both services can be implemented in our distributed system architecture. Requests must be delivered by atomic broadcast to ensure that all servers return the same answers. Updates to the database must be treated in the same way. The digital signature scheme of the service is replaced by the corresponding threshold signature scheme, which requires minimal changes to the clients in the case of [59]. In the server code, computing the digital signature is replaced by generating a signature share.

Note that atomic broadcast is crucial for delivering any request that changes the global state; only if a CA never changes its policy and all of its certificates are independent of each other, it suffices to use reliable broadcast.

4.4 Authentication Service

An *authentication service* verifies the claimed identity of a user or of a process acting on behalf of a user. The user must present secret information that identifies her or carry out a zero-knowledge identification protocol. If verification succeeds, the service will take some action to grant the request, like establish a session or return a cryptographic token for later use; this depends on the context in which the service is used. If the answer contains a freshly generated, random session key, as in of Kerberos [61], such an authentication server is also called a key distribution center (KDC). Communication between the authentication service and clients may be encrypted and signed with the public key of the service.

The security assumption about the authentication service is that it acts honestly when verifying a password or an identification protocol and never grants a request without having seen the proper identification. The reference data against which the verification occurs is assumed to be public but immutable; this is the case for Unix-style password authentication and for zero-knowledge identification protocols, for instance. But a KDC based on symmetric-key cryptography must also protect the corresponding master secret key.

A distributed authentication service consists of several authentication servers that are initialized by a trusted dealer and have access to the public reference data. Client requests are distributed by atomic broadcast to all servers. If the request contains sufficient information to authorize the user, the service must produce a suitable cryptographic token.

We distinguish two cases for this, depending on whether the cryptographic token uses public-key techniques or not:

- In a *public-key scenario*, the response of the authentication service is a digitally signed message, which can be thought of as a specialized certificate. Thus, the threshold signature protocols for a CA are used as described in Section 4.3.
- If *symmetric-key cryptography* is used, the servers maintain a shared master key and the response consists of an encryption under the master key, just as in Kerberos. An efficient non-interactive protocol to realize a distributed KDC was presented by Naor, Pinkas, and Reingold [50]. The cryptographic mechanism underlying their protocol is in fact the same as used for realizing the distributed common coin in our Byzantine agreement protocol [12] and integrates nicely with our architecture.

As in the general approach, a client can assemble the cryptographic token from the answers of all servers that authorizes her. Existing authorization protocols that use this token require some minimal changes in the cryptographic algorithms.

5 Extensions

We mention some extensions and improvements of our architecture. Although we have strived for a secure and fault-tolerant system in the given environment, the security could be strengthened by using "proactive" protocols, allowing for dynamic group changes, or using hybrid failure structures (not to be confused with generalized ones). Our atomic broadcast protocols involve a considerable overhead, in particular for large n, because security has been our primary design principle. Among the various possible optimizations, it seems most promising to design "optimistic" protocols, which run very fast if no corruptions occur but may fall back to slower protocols if necessary.

Proactive Protocols. Proactive security is a method to protect threshold-cryptographic schemes against a mobile adversary that can corrupt all parties during the lifetime of the system, but never more than t at once (see [13] for a survey). Proactive protocols divide time into epochs. All parties "reshare" their cryptographic secret keys between two epochs and delete all old key material. The model assumes an external mechanism for detecting corruptions and "cleaning up" a party. Because all secrets that the adversary has seen in the past become useless by resharing, the adversary never knows enough secret information to compromise the whole system.

Proactively secure protocols for our asynchronous system model are currently not known. One issue to be addressed first is how to integrate epochs into the asynchronous system model.

Dynamic Groups. The static nature of our system model may pose a problem for practical systems. Real systems evolve over time and grow or shrink together with the organizations that use them. Every such change would require a fresh setup of the complete system by a trusted dealer. Ideally, a system should reconfigure itself and dynamically increase or decrease the group size and the thresholds. However, special care is needed to ensure the safety of all keys during the changes; thus, at least some resharing of keys will be needed as in proactive protocols.

Note that similar motivation has led to the important idea of view-based group communication systems that tolerate crash failures. But dynamically changing the group seems much harder in the Byzantine model when cryptographic secrets are involved; this is currently an open problem. The recent work of Alvisi et al. [1] considers dynamic changes for Byzantine quorum systems, but their model does not involve confidentiality or any cryptographic techniques.

Hybrid Failure Structures. Another interesting direction is to treat crash failures separately from corruptions and adapt the protocols to such hybrid failure structures. After all, crashes are more likely to occur than intrusions and they are much easier to handle than Byzantine corruptions. For coping with transient server outages, the crash-recovery model seems also plausible (see references in [36]). Protocols in hybrid failure models have been investigated before [31, 49] so that we expect this to be feasible.

Optimistic Protocols. Optimistic protocols run very fast if no malicious adversary is at work and all messages are delivered promptly. If a problem is detected (typically because liveness is violated), they may switch into a more secure mode using protocols that guarantee progress. This idea is quite common in the literature [52, 15]. In our Byzantine context, one has to make sure that safety is never violated, though.

Bibliography

[1] L. Alvisi, D. Malkhi, E. Pierce, M. K. Reiter, and R. N. Wright, "Dynamic Byzantine quorum systems," in *Proc. International Conference on Dependable Systems and Networks (FTCS-30/DCCA-8)*, pp. 283–292, 2000.

[2] N. Asokan, V. Shoup, and M. Waidner, "Optimistic fair exchange of digital signatures," *IEEE Journal on Selected Areas in Communications*, vol. 18, pp. 591–610, Apr. 2000.

[3] R. Baldoni, J.-M. Helary, and M. Raynal, "From crash fault-tolerance to arbitrary-fault tolerance: Towards a modular approach," in *Proc. International Conference on Dependable Systems and Networks (FTCS-30/DCCA-8)*, pp. 273–282, 2000.

[4] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *Proc. 1st ACM Conference on Computer and Communications Security*, 1993.

[5] M. Ben-Or, "Another advantage of free choice: Completely asynchronous agreement protocols," in *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC)*, 1983.

[6] M. Ben-Or, R. Canetti, and O. Goldreich, "Asynchronous secure computation," in *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, 1993.

[7] M. Ben-Or, B. Kelmer, and T. Rabin, "Asynchronous secure computation with optimal resilience," in *Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC)*, 1994.

[8] J. Benaloh and J. Leichter, "Generalized secret sharing and monotone functions," in *Advances in Cryptology: CRYPTO '88* (S. Goldwasser, ed.), vol. 403 of *Lecture Notes in Computer Science*, pp. 27–35, Springer, 1990.

[9] D. Boneh and M. Franklin, "Efficient generation of shared RSA keys," in *Advances in Cryptology: CRYPTO '97* (B. Kaliski, ed.), vol. 1233 of *Lecture Notes in Computer Science*, pp. 425–439, Springer, 1997.

[10] G. Bracha, "An asynchronous $[(n-1)/3]$-resilient consensus protocol," in *Proc. 3rd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 154–162, 1984.

[11] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM*, vol. 32, pp. 824–840, Oct. 1985.

[12] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography," in *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 123–132, 2000. Full version available from Cryptology ePrint Archive, Report 2000/034, `http://eprint.iacr.org/`.

[13] R. Canetti, R. Gennaro, A. Herzberg, and D. Naor, "Proactive security: Long-term protection against break-ins," *RSA Laboratories' CryptoBytes*, vol. 3, no. 1, 1997.

[14] R. Canetti and T. Rabin, "Fast asynchronous Byzantine agreement with optimal resilience," in *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 42–51, 1993.

[15] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. Third Symp. Operating Systems Design and Implementation (OSDI)*, 1999.

[16] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[17] S. Chaudhuri, "More choices allow more faults: Set consensus problems in totally asynchronous systems," *Information and Computation*, vol. 105, no. 1, pp. 132–158, 1993.

[18] B. Chor and C. Dwork, "Randomization in Byzantine agreement," in *Randomness and Computation* (S. Micali, ed.), vol. 5 of *Advances in Computing Research*, pp. 443–497, JAI Press, 1989.

[19] R. Cramer, I. B. Damgård, and U. Maurer, "General secure multi-party computation from any linear secret sharing scheme," in *Advances in Cryptology: EUROCRYPT 2000* (B. Preneel, ed.), vol. 1087 of *Lecture Notes in Computer Science*, Springer, 2000.

[20] Y. Desmedt, "Threshold cryptography," *European Transactions on Telecommunications*, vol. 5, no. 4, pp. 449–457, 1994.

[21] Y. Deswarte, L. Blain, and J.-C. Fabre, "Intrusion tolerance in distributed computing systems," in *Proc. 12th IEEE Symposium on Security & Privacy*, pp. 110–121, 1991.

[22] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, pp. 644–654, Nov. 1976.

[23] D. Dolev, C. Dwork, and M. Naor, "Non-malleable cryptography," *SIAM Journal on Computing*, vol. 30, no. 2, pp. 391–437, 2000.

[24] A. Doudou, B. Garbinato, and R. Guerraoui, "Abstractions for devising Byzantine-resilient state machine replication," in *Proc. 19th Symposium on Reliable Distributed Systems (SRDS 2000)*, pp. 144–152, 2000.

[25] D. E. Eastlake, "RFC 2535: Domain name sytstem security extensions," Mar. 1999.

[26] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," in *Foundations of Computation Theory* (M. Karpinsky, ed.), vol. 158 of *Lecture Notes in Computer Science*, Springer, 1983. Also published as Tech. Report YALEU/DCS/TR-273, Department of Computer Science, Yale University.

[27] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, pp. 374–382, Apr. 1985.

[28] M. Fitzi and U. Maurer, "Efficient Byzantine agreement secure against general adversaries," in *Proc. 12th International Symposium on Distributed Computing (DISC)*, vol. 1499 of *Lecture Notes in Computer Science*, pp. 134–148, Springer, 1998.

[29] Y. Frankel, P. MacKenzie, and M. Yung, "Robust efficient distributed rsa key generation," in *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 663–672, 1998.

[30] J. A. Garay, R. Gennaro, C. Jutla, and T. Rabin, "Secure distributed storage and retrieval." Proc. 11th International Workshop on Distributed Algorithms (WDAG), 1997.

[31] J. A. Garay and K. J. Perry, "A continuum of failure models for distributed computing," in *Proc. 6th International Workshop on Distributed Algorithms (WDAG)*, 1992.

[32] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure key generation for discrete-log based cryptosystems," in *Advances in Cryptology: EUROCRYPT '99* (J. Stern, ed.), vol. 1592 of *Lecture Notes in Computer Science*, pp. 295–310, Springer, 1999.

[33] O. Goldreich, *Foundations of Cryptography: Basic Tools.* Cambridge University Press, 2001. To appear.

[34] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *Journal of the ACM*, vol. 33, pp. 792–807, Oct. 1986.

[35] S. Goldwasser, S. Micali, and R. L. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM Journal on Computing*, vol. 17, pp. 281–308, Apr. 1988.

[36] R. Guerraoui, M. Hurfin, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper, "Consensus in asynchronous distributed systems: A concise guided tour," in *Advances in Distributed Systems* (S. Krakowiak and S. Shrivastava, eds.), vol. 1752 of *Lecture Notes in Computer Science*, pp. 33–47, Springer, 2000.

[37] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems* (S. J. Mullender, ed.), New York: ACM Press & Addison-Wesley, 1993. An expanded version appears as Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca NY, 1994.

[38] M. Hirt and U. Maurer, "Player simulation and general adversary structures in perfect multi-party computation," *Journal of Cryptology*, vol. 13, no. 1, pp. 31–60, 2000.

[39] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "The SecureRing protocols for securing group communication," in *Proc. 31st Hawaii International Conference on System Sciences*, pp. 317–326, IEEE, Jan. 1998.

[40] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978.

[41] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, July 1982.

[42] N. Lynch and M. R. Tuttle, "Hierarchical correctness proofs for distributed algorithms," in *Proc. 6th ACM Symposium on Principles of Distributed Computing (PODC)*, 1987.

[43] N. A. Lynch, *Distributed Algorithms.* San Francisco: Morgan Kaufmann, 1996.

[44] N. A. Lynch and M. R. Tuttle, "An introduction to input/output automata," *CWI Quaterly*, vol. 2, pp. 219–246, Sept. 1989.

[45] D. Malkhi, M. Merritt, and O. Rodeh, "Secure reliable multicast protocols in a WAN," *Distributed Computing*, vol. 13, no. 1, pp. 19–28, 2000.

[46] D. Malkhi and M. K. Reiter, "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.

[47] D. Malkhi and M. K. Reiter, "An architecture for survivable coordination in large distributed systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 187–202, 2000.

[48] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography.* Boca Raton, FL: CRC Press, 1997.

[49] L. E. Moser and P. M. Melliar-Smith, "Byzantine-resistant total ordering algorithms," *Information and Computation*, vol. 150, pp. 75–111, 1999.

[50] M. Naor, B. Pinkas, and O. Reingold, "Distributed pseudo-random functions and KDCs," in *Advances in Cryptology: EUROCRYPT '99* (J. Stern, ed.), vol. 1592 of *Lecture Notes in Computer Science*, Springer, 1999.

[51] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM*, vol. 27, pp. 228–234, Apr. 1980.

[52] F. Pedone and A. Schiper, "Optimistic atomic broadcast," in *Proc. 12th International Symposium on Distributed Computing (DISC)*, 1998.

[53] D. Powell (Guest Ed.), "Group communication," *Communications of the ACM*, vol. 39, pp. 50–97, Apr. 1996.

[54] M. Reiter, "Secure agreement protocols: Reliable and atomic group multicast in Rampart," in *Proc. 2nd ACM Conference on Computer and Communications Security*, 1994.

[55] M. K. Reiter, "Distributing trust with the Rampart toolkit," *Communications of the ACM*, vol. 39, pp. 71–74, Apr. 1996.

[56] M. K. Reiter and K. P. Birman, "How to securely replicate services," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 986–1009, May 1994.

[57] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, Feb. 1978.

[58] V. Shoup, "Why chosen ciphertext security matters," Research Report RZ 3076, IBM Research, Nov. 1998.

[59] V. Shoup, "Practical threshold signatures," in *Advances in Cryptology: EUROCRYPT 2000* (B. Preneel, ed.), vol. 1087 of *Lecture Notes in Computer Science*, pp. 207–220, Springer, 2000.

[60] V. Shoup and R. Gennaro, "Securing threshold cryptosystems against chosen ciphertext attack," in *Advances in Cryptology: EUROCRYPT '98* (K. Nyberg, ed.), vol. 1403 of *Lecture Notes in Computer Science*, Springer, 1998.

[61] J. G. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: An authentication service for open network systems," in *Usenix Conference Proceedings*, pp. 191–202, Mar. 1988.

[62] D. R. Stinson, *Cryptography: Theory and Practice.* CRC Press, 1995.

[63] P. Veríssimo, A. Casimiro, and C. Fetzer, "The timely computing base: Timely actions in the presence of uncertain timeliness," in *Proc. International Conference on Dependable Systems and Networks (FTCS-30/DCCA-8)*, pp. 533–542, 2000.

[64] P. Veríssimo and N. F. Neves, eds., *Service and Protocol Architecture for the MAFTIA Middleware*. Deliverable D23, Project MAFTIA IST-1999-11583, Jan. 2001.